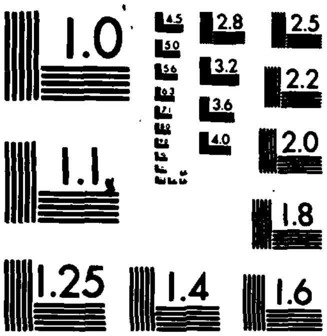
AD-A137 687 DISTRIBUTED PROCESSING TOOLS DEFINITION VOLUME 2 1/2
APPLICATION OF SOFTWARE..(U) GENERAL DYNAMICS FORT
WORTH TX DATA SYSTEMS DIV H C CONN ET AL. JUN 83
UNCLASSIFIED RADC-TR-83-107-VOL-2 F30602-81-C-0142 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

RADC-TR-83-107, Vol II (of three)
Final Technical Report
June 1983

DISTRIBUTED PROCESSING TOOLS DEFINITION Application of Software Engineering Technology

General Dynamics Corporation

Herbert C. Conn, Jr.; David L. Kellogg; Rodney M. Bond; States L. Nelson; Scott L. Harmon; Sue A. Johnson; William D. Baker and Paul B. Dobbs

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-107, Volume II (of three) has been reviewed and is approved for publication.

APPROVED: Lawrence M. Lombardo.

LAWRENCE M. LOMBARDO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
RADC-TR-83-107, Vol II (of three)

2. GOVT ACCESSION NO.
AD-A137 687

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
DISTRIBUTED PROCESSING TOOLS DEFINITION
Application of Software Engineering Technology

5. TYPE OF REPORT & PERIOD COVERED
Final Technical Report
Jun 81 - Jan 83

6. PERFORMING ORG. REPORT NUMBER
N/A

7. AUTHOR(s)
H.C. Conn, Jr. S.L. Nelson W.D. Baker
D.L. Kellogg S.L. Harmon P.B. Dobbs
R.M. Bond S.A. Johnson

8. CONTRACT OR GRANT NUMBER(s)
F30602-81-C-0142

9. PERFORMING ORGANIZATION NAME AND ADDRESS
General Dynamics Corporation
Data Systems Division
P O Box 748, Fort Worth TX 76101

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
62702F
55811829

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (COEE)
Griffiss AFB NY 13441

12. REPORT DATE
June 1983

13. NUMBER OF PAGES
178

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Same

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Lawrence M. Lombardo (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Computer Embedded Distributed Processing Systems
Computer Software Technologies
Computer Hardware Technologies
Computer Object Oriented Modularization
Computer System Life Cycle Phase Support

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The objective of this three-phase effort is to (1) Identify the hardware/software technology pertinent to the implementation of tightly-coupled embedded distributed systems for DoD applications, (2) Establish an integrated approach regarding the total life-cycle software development period with correlation as to the applicability of existing/near-term software engineering methodology, techniques and tools to each life-cycle phase, and (3) Define the functional design requirements pertinent to the far-term development of needed software engineering methodology, techniques and tools. A product of this effort is the recommended design of a system support environment encompassing the integrated implementation of candidate software engineering tools.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

1.0 Technical Report Summary

General Dynamics Data Systems Division is under contract to Rome Air Development Center to conduct a study entitled Distributed Processing Tools Definition. The objectives are to investigate the requirements for software life cycle support in embedded distributed processing systems and to specify applicable software tools and techniques by life cycle phase.

1.1 Project Overview

The study is divided into three phases which are illustrated in Figure 1-1. Phases I and II of the study have been completed, and their results are described in the present Technical Report.

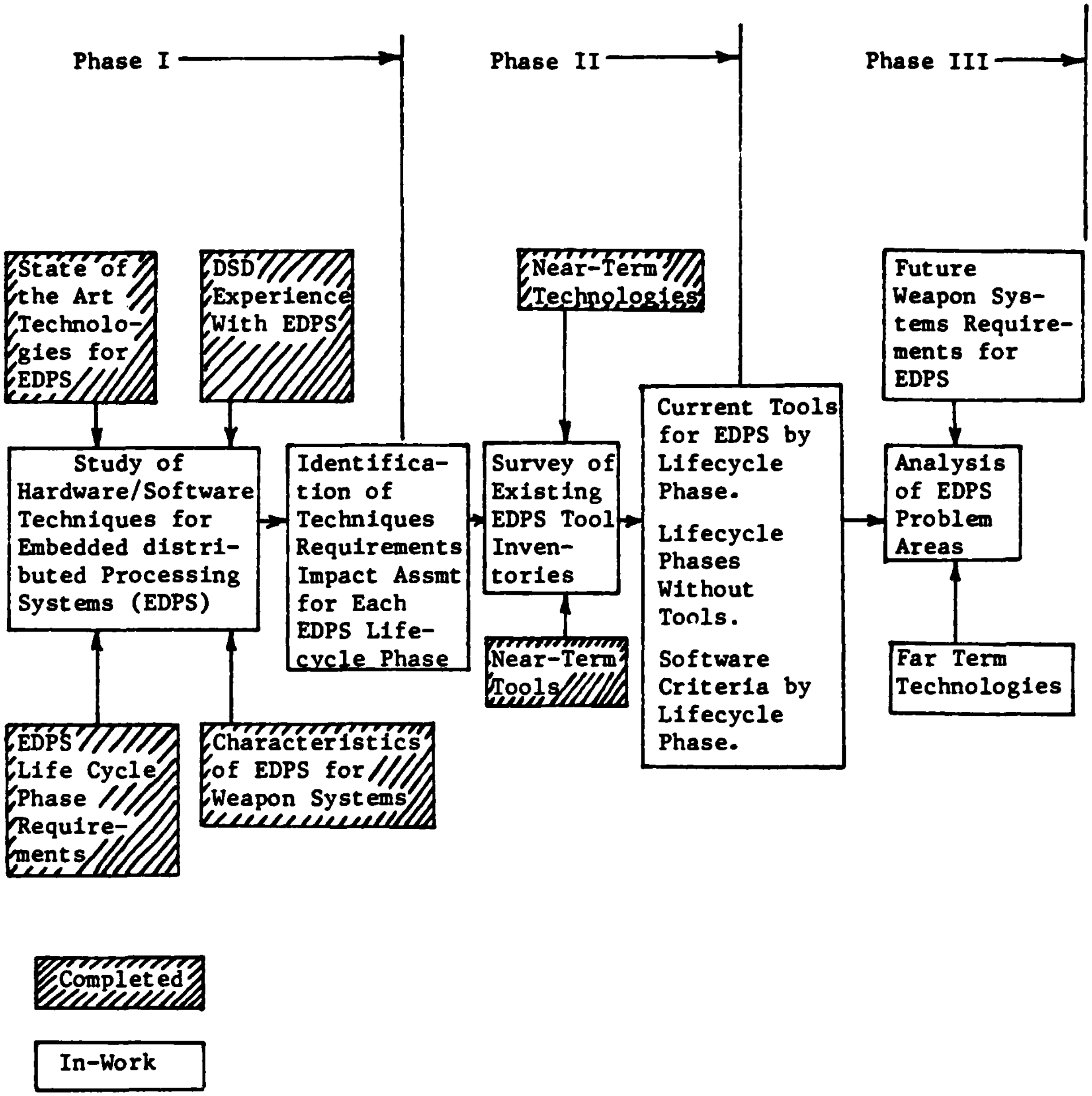

Phase I
Phase II
Phase III
State of the Art Technologies for EDPS
DSD Experience With EDPS
Near-Term Technologies
Future Weapon Systems Requirements for EDPS
Current Tools for EDPS by Lifecycle Phase.
Lifecycle Phases Without Tools.
Software Criteria by Lifecycle Phase.
Study of Hardware/Software Techniques for Embedded distributed Processing Systems (EDPS)
Identification of Techniques Requirements Impact Assmt for Each EDPS Lifecycle Phase
Survey of Existing EDPS Tool Inventories
Analysis of EDPS Problem Areas
Near-Term Tools
Far Term Technologies
EDPS Life Cycle Phase Requirements
Characteristics of EDPS for Weapon Systems
Completed
In-Work

Figure 1-1 Overview of the Distributed Processing Tools Definition Study

1.2 Phase II Conclusions

The principal technical perspective used in the following conclusions assumes a combined system functionality for hardware and software. Embedded distributed processing systems require both to be operable at the same time. Each conclusion is followed by a reference to its appropriate discussion in the Technical Report.

1) Development of Distributed Processing Systems is best supported by an Integrated Software Support Environment (ISSE) (see paragraph 3.0).

2) It is economically efficient to make ISSEs for distributed processing open-ended (see paragraph 3.0).

3) Efficient static analysis of deterministic systems is possible (see paragraph 3.1).

4) Further research is needed on static analysis of non-deterministic systems (see paragraph 3.1).

5) Ada will require a cross-reference tool with extensive capabilities (see paragraph 3.1.1).

6) Ada will require static analyzers for its concurrency constructs (see paragraph 3.1.1).

7) The requirements and design phases must be heavily stressed and further automated to build cost efficient and reliable distributed systems (see paragraph 3.2).

8) Rapid prototypers and simulators are necessary support for distributed processing system designers (see paragraph 3.2).

9) An automated requirements tool consisting of a requirements interface processor, a requirements document generator, a requirements analyzer, and a requirements language translator has proven to be useful during the requirements phase and should be further developed for distributed systems (see paragraph 4.2.1).

10) The design phase is virtually poverty-stricken with regard to tools. The only tools available currently in this area are program design language processors, even though this phase is one of the most important in determining the overall quality of the finished system (see paragraph 4.2.2).

11) The tools of the coding phase can be divided into two categories: tools dealing with languages and tools dealing with standards. The tools dealing with languages exist for any usable high level language, with the possible exception of checkout compilers. Since the functions of a checkout compiler overlap both the coding and the testing phase, checkout compilers should be developed where possible. Standards auditors and formatters should be developed to enhance consistency of code (see paragraph 4.2.3).

12) The tools of the testing phase are concerned with checking for adherence to requirements and design and resolving coding errors. Referring to Table 4-2, all needed criteria for the life cycle phase of testing (Table 4-1) have been enhanced. Due to the applicability of the testing tools to other phases, criteria not identified as needed have been enhanced in the testing phase. Testing has an abundance of tools; however, tools should be more concerned with distributedness of the software (see paragraph 4.2.4).

13) The tools of the maintenance phase consist of a version generator, rapid reconfiguration, and report generator. Many criteria are not addressed in the maintenance phase (Table 4-2) according to the

prescribed criteria in Table 4-1. More tools should be developed to satisfy the requirements since it is the phase ensuring the continuation of quality throughout the life of the software (see paragraph 4.2.5).

14) The operations phase has few tools since it is usually a user phase with little software engineering activity; however, tools should be developed to ensure operational quality of the system. Comparing Table 4-2 with the required criteria for the operations phase in Table 4-1, the diagnostic analyzer and system builder enhance less than one-half of the criteria. Criteria such as control of data access and ease of use, which are critical user criteria, need to be addressed by new tool development (see paragraph 4.2.6).

15) Knowledge-based systems will be components of far-term artificial intelligence techniques both in development support systems and in operational application systems. Near-term tool definition should take into account the potential for integration into future highly intelligent systems (see paragraph 3.2.4).

2.0 Scope of Embedded Distributed Processing Systems

Distributed processing systems can be characterized by their position within a three-dimensional space. Each axis of that space can be used to separate the non-distributed environment from its distributed counterpart. The first of these axes concerns the distribution of hardware. It ranges from a single central processor unit to multiple computers. The single central processor unit is characterized by one control unit, one arithmetic logic unit, and one central memory. The multiple computers are characterized by multiple general purpose central processing units which have their own control units, arithmetic logic units, central memories, and input/output systems. Several configurations lie somewhere between these two extremes. In summary, the five following generic hardware configurations have been isolated and are in common use.

1) Single central processor unit characterized by
    - i) one control unit
    - ii) one arithmetic logic unit and
    - iii) one central memory.
2) Multiple execution units characterized by
    - i) one control unit
    - ii) mutiple, identical arithmetic logic units and

iii) possibly multiple, independent central memories.

3) Separate "specialized" functional units characterized by

i) one general purpose control unit and

ii) multiple arithmetic logic units or processing units

a) some may be specialized units

b) each is limited and

c) all may be identical general purpose units.

4) Multiple processors characterized by

i) multiple control units

ii) multiple arithmetic logic units

iii) possibly multiple, independent, central memories and

iv) a single, coordinated input/output system.

5) Multiple computers characterized by

i) multiple general purpose central processing units with their own control unit, arithmetic logic unit, central memory, and input/output system.

Distributed processing systems are usually composed of multiple processors and multiple computers which are the fourth and fifth generic hardware configurations. The non-distributed environment is typified by the single central processor unit and multiple execution units which are the first and second generic configurations.

The second axis within the three-dimensional space concerns the distribution of control. It ranges from a single, fixed point of control to multiple control points which are not necessarily homogeneous but which cooperate on the execution of a task. In all there are seven categories of generic control:

1) Single, fixed control point;
2) Fixed master-slave relationship;
3) Dynamic master-slave relationship;
4) Multiple points of control which are totally autonomous;
5) Multiple points of control which cooperate on the execution of a task which has been subdivided into sub-tasks;
6) Replicated, identical points of control cooperating on the execution of a task; and
7) Multiple control points which are not necessarily homogeneous but which cooperate on the execution of a task.

Distributed processing systems are usually typified by the top three categories of generic control, i.e., multiple points of control, replicated points of control, and multiple control points. Conversely, non-distributed processing systems are characterized by the bottom two categories, i.e., single control points and fixed master-slave reltionships.

After an examination of only two of the three dimensions, several obvious observations can be made. First, the movement within the marketplace has been and will continue to be from single processors to multiple processors. As these multiple processors become more prevalent, non-distributed control policies become questionable. Second, as individual nodes reach parity within multiple processor configurations, choosing one node over another for control purposes is increasingly difficult to defend. Third, configurations which run without an overriding executive are now possible. In summary, the multiple processor configurations have complicated the present situation.

The third and final axis within the three-dimensional space concerns the distribution of data bases. In terms of complexity it ranges from a single copy data in secondary storage to a completely partitioned data base without a master file or director. Many gradations exist between these two extremes. To date approximately seven generic categories of distributed data bases have been isolated. The first three are associated with centralized data bases while the last four are associated with the emerging distributed data bases.

Centralized data bases include:

1) single copy, secondary storage;
2) single copy, primary memory; and
3) distributed files with single, centralized directories.

Distributed data bases include:

4) replicated data bases;
5) partitioned data base with a complete master copy;
6) partitioned data base with a master directory; and
7) partitioned data base without a master file or directory.

File structures must not only accommodate the serialized requests from a single processor but must also accommodate the concurrent requests from several processors.

When all three axes of the three-dimensional space are considered, the spatial location of distributed processing is clearly different from non-distributed or centralized processing. That difference is illustrated in Figure 2-1. Such distributed processing systems are usually components within larger systems, i.e., they are embedded.

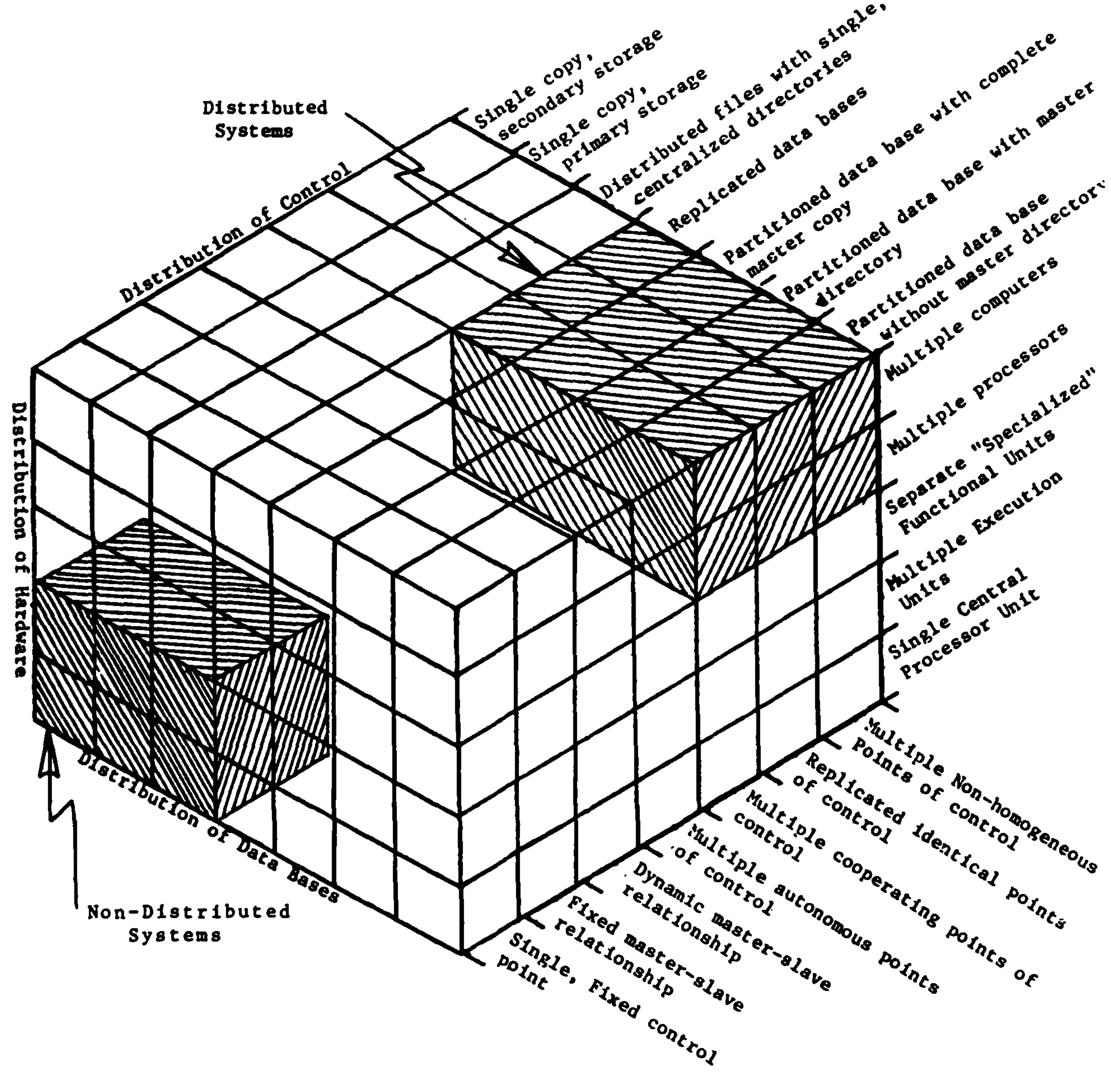

Distributed Systems
Distribution of Control
Single copy, secondary storage
Single copy, primary storage
Distributed files with single, centralized directories
Replicated data bases
Partitioned data base with complete master copy
Partitioned data base with master directory
Partitioned data base without master directory
Multiple computers
Multiple processors
Separate "Specialized" Functional Units
Multiple Execution Units
Single Central Processor Unit
Multiple Non-homogeneous Points of control
Replicated identical points of control
Multiple cooperating points of control
Multiple autonomous points of control
Dynamic master-slave relationship
Fixed master-slave relationship
Single, Fixed control point
Distribution of Hardware
Distribution of Data Bases
Non-Distributed Systems

Figure 2-1 Characterization of Distributed Processing Systems Within the Three-Dimensional Space Comprised of the Following Axes: Distribution of Hardware, Distribution of Controls and Distribution of Data Bases

The application functions performed by these larger systems commonly take precedence over their embedded components. Examples would include large mainframes composed of tightly-coupled multiple processors. The users of such mainframes are seldom aware of their embedded components and assume a centralized environment. Of course the tools available to such users are designed to operate in a centralized environment. The capability of a tightly-coupled, concurrently-operated environment is seldom made available to users because of a shortage of tools. Despite this shortage, decision making concerning embedded distributed processing systems requires mastery of data movement and a solid understanding of computational efficiency. Enough information about the distribution of hardware, control, and data bases must be known within appropriate real-time constraints in order to reach informed decisions. The functionality of the final product depends upon these informed decisions.

Because of inter-relations between axes, the analogy of a three-dimensional space can only be carried so far. Hardware, control, and data bases are not independent of one another in a distributed processing environment. A change in one impacts the other.

Growth in data bases can and does force change in tightly-coupled hardware to accommodate increased information flow. As tightly-coupled hardware changes, the control process is impacted. The bottom line remains functionality despite what happens to either the data base, hardware, or control. That functionality represents a massive tradeoff between hardware and software. Furthermore, each one of these tradeoffs can be characterized by its position within the inter-related, three-dimensional space.

2.1 Increased Distribution of Hardware

As distributed processing systems are winning wide acceptance within the marketplace, their ability to move and efficiently process large amounts of data has attracted military interest. Current military computer systems span the spectrum from "smart bombs" and bullets to global communications systems. The resultant military applications fall into two very large generic classifications:

1) Weapon systems and
2) Communications systems

Included within weapon systems are armament, aeronautical, missile, and space applications. Included within communications systems are command/control/communications applications as well as mission and force management functions. Each category occupies a characteristic position within the three-dimensional space whose axes are the distribution of control, the distribution of hardware, and the distribution of data bases. These positions are presented in Figure 2-2.

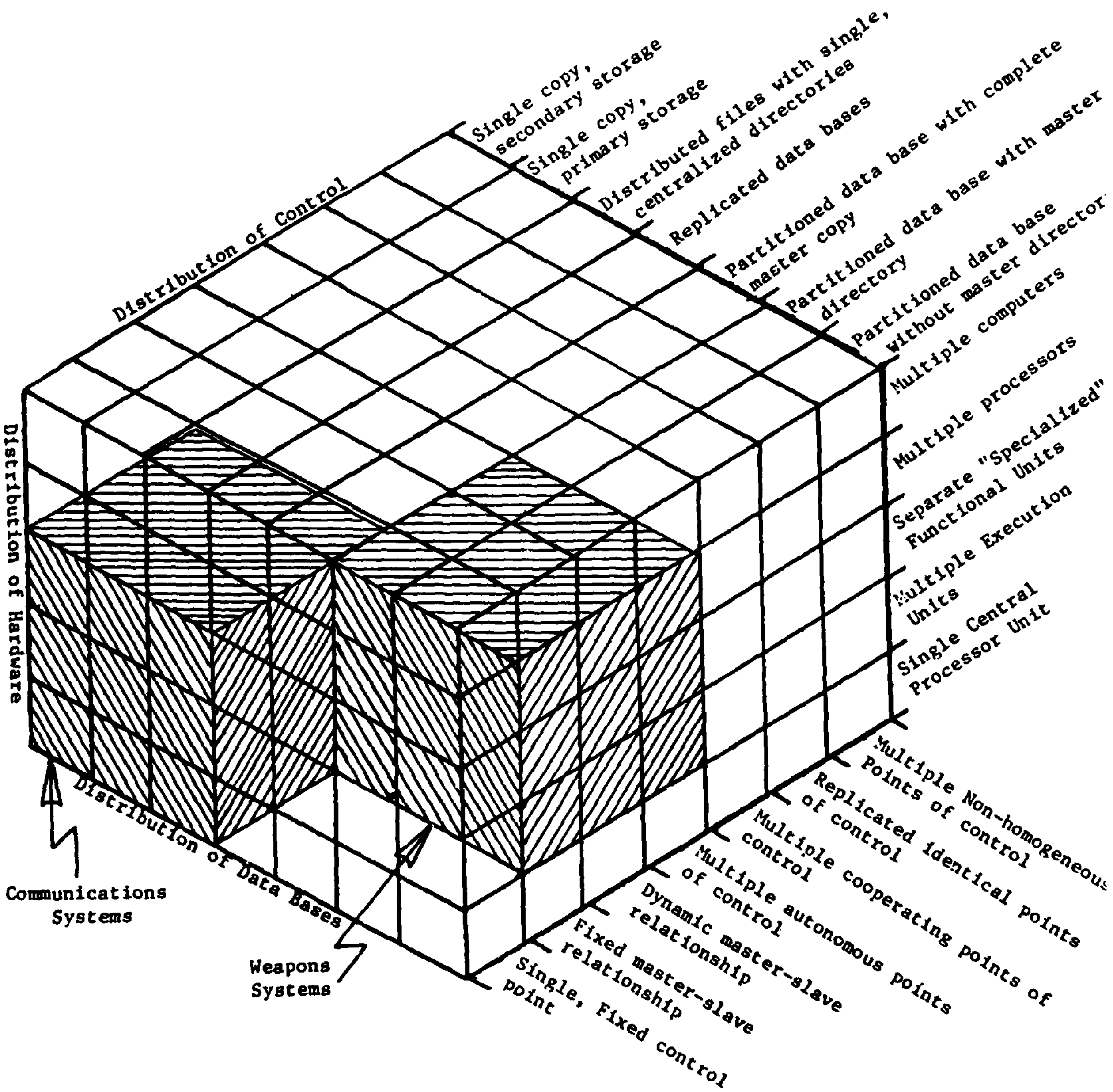

Distribution of Control
Single copy, secondary storage
Single copy, primary storage
Distributed files with single, centralized directories
Replicated data bases
Partitioned data base with complete master copy
Partitioned data base with master directory
Partitioned data base without master director
Distribution of Hardware
Multiple computers
Multiple processors
Separate "Specialized" Functional Units
Multiple Execution Units
Single Central Processor Unit
Distribution of Data Bases
Multiple Non-homogeneous Points of control
Replicated identical points of control
Multiple cooperating points of control
Multiple autonomous points of control
Dynamic master-slave relationship
Fixed master-slave relationship
Single, Fixed control point
Communications Systems
Weapons Systems

Figure 2-2 Characterization of Weapon Systems and Communications Systems Within the Three-Dimensional Space Comprised of the Following Axes: Distribution of Hardware, Distribution of Control, and Distribution of Data Bases

Currently the communications systems contain the characteristics of non-distributed systems. However, growth is occurring and such systems are expanding their capability. Dynamic master-slave relationships and separate "specialized" functional units are beginning to appear. Furthermore, distributed files with single, centralized directories are beginning to appear in some leading-edge communications systems. Notably the connections remain loosely-coupled and are exemplified by command/control/communications systems. Such loose-coupling results from the constrained bandwidth of the connection technologies currently in use. However, these technologies are undergoing great change. Bandwidths are increasing and as they *increase*, the capability for tighter-coupling is enabled. Possibly replicated data bases with multiple processors will be available in the near term.

Currently weapon systems lie somewhere between the non-distributed and distributed technologies. Their connections are more tightly-coupled than the communications systems. Although their bandwidths are usually greater than communications systems, they are not as high as the distributed systems. As a consequence, some of the capability of distributed systems is not available in current weapon systems. The reason lies in the control functions. By their nature, weapon systems must be tightly controlled. A stores management system may inventory, fire, and update a particular weapon system's status. However, that same system is not fail-safe and must verify its action beforehand through the concurrence of a control system. Its actual operation is characteristic of an embedded system, not a completely distributed system. In the near term, weapon systems will retain their present orientation. Completely fail-safe operation of distributed systems remains a concept for future implementation.

2.1.1 Weapon Systems

Advanced weapon systems are becoming increasingly complex and rely upon computers and embedded processors to operate. One example is provided by self-contained surface mobile weapon systems used by the Army. Such systems rely heavily upon an internal fire control computer and an embedded navigation system to provide operational direction. As newer systems evolve, an increasing number of remote functions are being incorporated. One such function is a remote operating console which requires more tightly-coupled data bases and a high level of data exchange. Traditional methods of fault detection and isolation no longer apply in such configurations.

Failures can occur and remain undetected simply because of sheer complexity. An example would be a minor logic error which occurred intermittently within the internal fire control computer. If a sufficient number of errors were generated, the weapon's accuracy would be destroyed. Since the problem is intermittent, routine maintenance would probably not perceive it.

Intermittent malfunctions are a problem for weapon systems in general, not surface mobile systems in particular. If these malfunctions occur at a critical time during an engagement, the effectiveness of a weapon system may be nullified. The internal fire control computer in a self-contained surface mobile weapon system illustrates how important a malfunction can be. Its embedded navigation system could have shifted modes from target search to target track when a malfunction caused the reverse shift. Instead of directing the system to the target, the fire control computer begins to search for that target. Observing such a phenomenon from the remote operating console does not alleviate the situation. To regain the lost target tracking mode, some sort of direct intervention must be initiated. Alternative modes of optical sighting and infrared detection schemes are often activated. In any case, the preferred action is to resolve and correct the malfunction source. This involves the direct detection and isolation of faults in real-time which is no trivial matter. Compounding the problem in our example is the implementation of a mode shift. Such a shift is related to the distribution of control. The embedded navigation system is capable of directing the fire control computer to the target but needs the concurrence of the fire control computer itself. This concurrence is a fail-safe mechanism to assure adequate operation of the surface mobile weapon system. In essence, the fire control computer concurs on the changes in operational modes. As illustrated, the ability of this computer to override the embedded navigation sytem can also create problems. What emerges is a need for new software tools.

Real-time fault isolation requirements are being satisfied in aircraft systems as well as spacecraft systems. The approach used by both systems is system redundancy through the use of multiple processors. When malfunctions occur, the redundant system automatically activities itself to maintain the operational mode and the same level of performance. The problem with such an approach is cost. The architecture in surface mobile weapon systems simply cannot afford the extra cost for redundancy. Consequently, software tools must be relied upon to accomplish the same objectives as system redundancy. These objectives are two in number:

1) provide real-time fault detection and advise a console operator of his alternatives and

2) achieve real-time fault isolation and retain specific failure parameters to diagnose the intermittent malfunctions.

As weapon systems have grown more complex, the verification of software has assumed greater importance. Missile systems provide a good example of evolving complexity. The transition from ballistic capability to cruise capability requires significantly greater amounts of software to be written and verified. General Dynamics has studied various software verification techniques and determined some of their shortcomings. In the context of weapon systems these shortcomings include computational overflows and program time constraints. Other methods of verification are required to address such problem areas, e.g., program testing based upon realistic simulations. Such realistic simulations can be used to verify more than computational overflows and program time constraints within weapon systems application software. Both software verification techniques and realistic simulations can be used to detect the following types of software errors:

1) Input conversion problems;

2) Output conversion problems;

3) Mathematical calculation problems;

4) Logic decision problems;

5) Path analysis problems;

6) Mathematical precision problems;

7) Lack of computational precision;

8) Initialization problems; and

9) Switches in operational modes.

Testing efforts based upon realistic simulations are clearly related to the distributions of hardware and control. The software verification techniques ignore the distribution of hardware but concentrate on the distributions of data bases and control.

As the complexity of weapon systems increases, the distribution of hardware also increases. The use of multiple processors has already been referenced. Such processors are usually tightly-coupled since they share common resources, e.g., the same data base. Viewed from the standpoint of statistics, such resource sharing increases the degrees of freedom over which a weapon system can operate. State of the art realistic simulations can now accommodate six degrees of freedom, i.e., six parameters can vary concurrently. The more de-centralized a system becomes, the greater its degrees of freedom. Extremely complex weapon systems require rigorous software testing.

If the actual operation of such weapon systems is to be avoided, realistic simulations and software verification techniques become extremely important. In a sense, the complexity of weapon systems will be constrained to our ability to verify their software or to simulate their subsequent operation realistically.

2.1.2 Communication Systems

Advanced communication systems are becoming increasingly complex and rely upon computers and embedded processors to operate. Such systems include command/control/communication applications as well as mission and force management functions. From the standpoint of users such systems seem to be loosely-coupled or completely uncoupled. From the standpoint of network designers, such systems are becoming tightly-coupled. Both viewpoints are valid although they seem to be contradictory. An explanation can be developed through an examination of Figure 2-3 concerning the nodes and links within a generic network system. Several levels of communication take place concurrently within such a system. The most essential layers concern the operation of the network itself. ᵀn the example this includes the network hosts X, Y, and Z. Their communication links comprise the backbone or trunk of the network. They incorporate the lowest three levels of the International Standards Organization Open System Interconnection Model devoted to the movement of bits, frames, and packets. These backbones or trunks support the next three layers of the Open System Model which, in turn, support the user node requirements for the transport, interaction, and presentation of information. In the context of user applications the communication appears to be loosely-coupled or completely uncoupled because of the operation of those presentation, session, or transport layers. However, the backbones or trunks operate in a tightly-coupled format, i.e., resources are shared between network nodes.

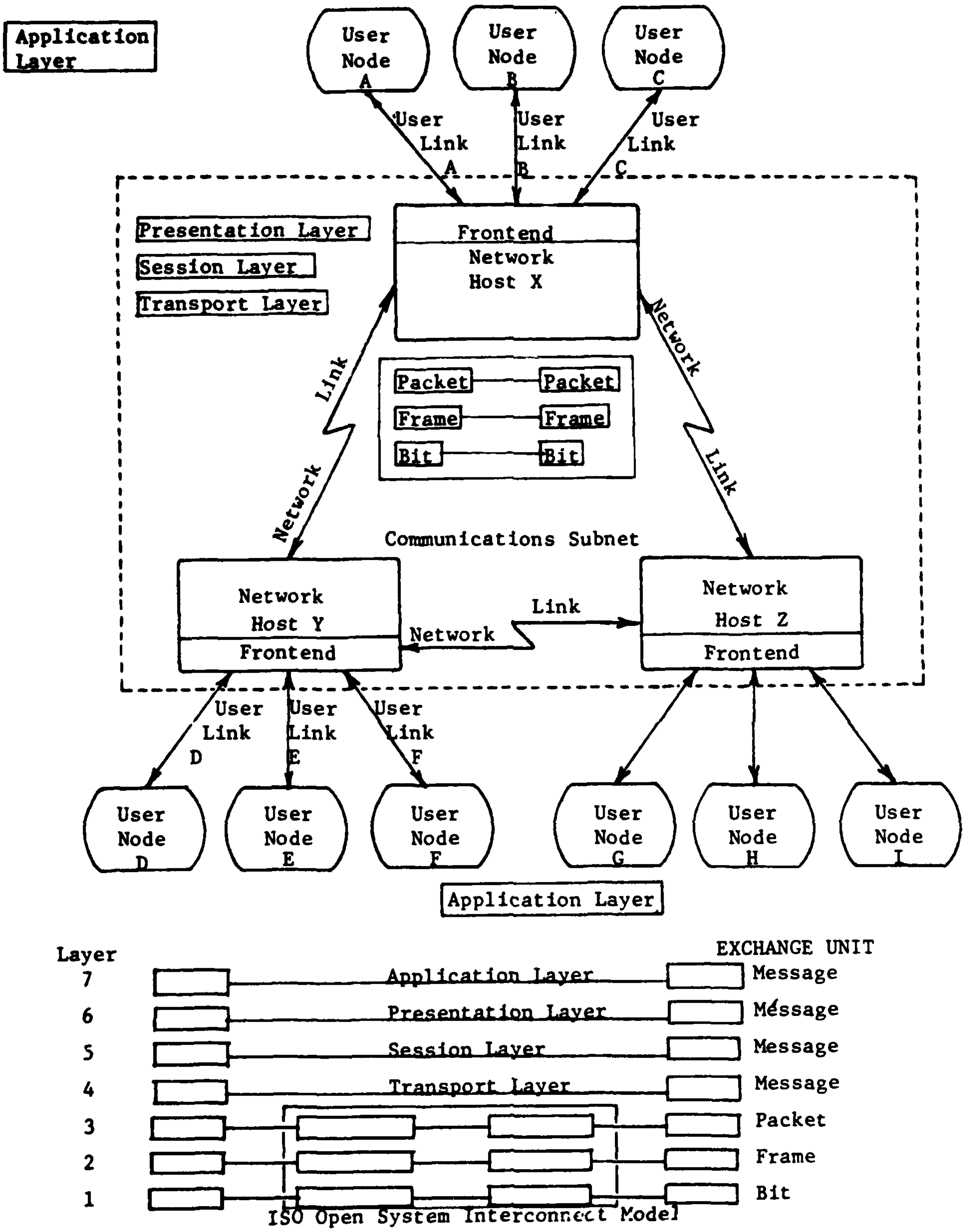


Figure 2-3 Classification of Nodes and Links Within a Network System

In the context of Figure 2-3, software is operating at several different locations. Each location addresses a particular layer or set of layers within the Open System Model. Growth within distributed systems can be measured in terms of increased computational power being placed at these strategic locations within networks. Starting from the standpoint of a user, several observations can be made concerning the software operating characteristics of various implementations of the Open System Interconnection Model. The first location at which significant computational power is encountered occurs at the network host level. Each host has its own front end to support a variety of user nodes. Two types of software are evident at this juncture. One resides upon each network host and addresses the computational function. The other resides within the front end of each network host and within the user nodes serviced by the front end of that particular host. It addresses the management functions of formatting and routing messages, nodal commands, transferring data, and updating status information. The differences between these two types of software are illustrated in the following table.

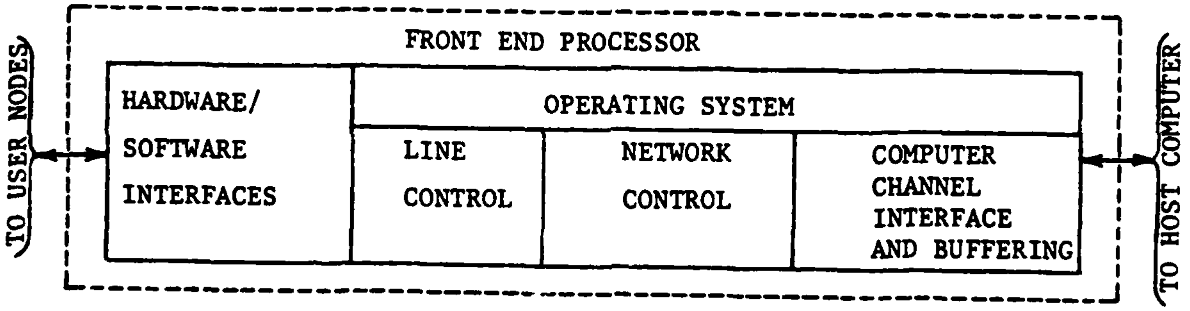
Network Host Front End and User Nodes Functions

Computational Functions	Management Functions
Requires data in binary format	Handles data in one of several transmission codes
Requires uncompressed data	Data compressed for efficient transmission
Processes complete blocks of data	Data transmitted in bit-by-bit serial format
Processes only data	Transmits data and line protocol information
Controls own timing	Handles a variety of timings dependent upon devices and operator speeds.

Obviously communication software between the network host and its user nodes must accommodate both the computational and management functions. It does so by addressing the problem at a variety of levels. The first is the physical interface itself. It concerns how the network host actually transfers the bits and frames between itself and the user nodes it services. These are the first two layers of the Open System Interconnection Model. The second level used to address the software problem concerns line control. It uses the bits and frames of the physical interface to direct the flow of information the network host and its user nodes. When software satisfies the physical interface and line control requirements, the first three layers of the Open System Interconnection Model have been implemented between the network host and its user nodes. These layers are called the communication subnet. A variety of such subnets are available within the communications marketplace, e.g., IEEE 802 interfaces and Ethernet. The third level used to address the communications software problem concerns control of the network. Whereas the previous two levels usually operate within loosely-coupled and uncoupled formats, network control usually does not. Such control addresses communication between individual network hosts. Within this level, resources are usually shared and software operates concurrently on the various network hosts. In effect, this level parallels the first level of the physical interface between an individual network host and its user nodes. The major difference is that this third level deals with the physical interface between network hosts themselves. When this latter interface operates, the network backbone or trunk is implemented. The efficiency of that backbone or trunk is a product of how the physical interface between network hosts is used. The software addressing that issue occurs within the fourth level of abstraction in the approach to communication software. At this level a network operating system resides. It distributes control, operates the various network hosts, and distributes the network databases. Obviously such software operates within a concurrent environment and is tightly coupled. In summary, the following four levels of abstraction are used to address the communication software between the network hosts and their respective user nodes.

1. Physical Interface (loosely-coupled or uncoupled)
2. Line Control (loosely-coupled or uncoupled)
3. Network Control (tightly-coupled and concurrent)
4. Operating System (tightly-coupled and concurrent)

The following diagram relates these levels of abstraction to one another.



Information flows between the individual network hosts and their respective user nodes via the user line interface. Such flow can be accomplished through hardware itself or software driving such hardware. Software drivers serve to transmit control information to user nodes and monitor traffic conditions. The software drivers at the line control level actual regulate those traffic conditions being monitored at the line interface level. They do not concern themselves with message content but simply concentrate on the movement of packets, frames, and bits.

Network control software performs the formatting function. It creates a single data stream and imposes a message structure upon the subsequent information flow between network hosts. This latter flow is managed by the network operating system which can range from single routine handling peripheral devices to very complex routines handling concurrent environments. The sophistication required of a network operating system is related to the hardware topology. Some of the most notable network topologies follow:

1. fully connected topology
2. generalized tree topology,
3. minimal spanning tree (MST) topology.
4. bus topology,

5. loop (or ring) topology

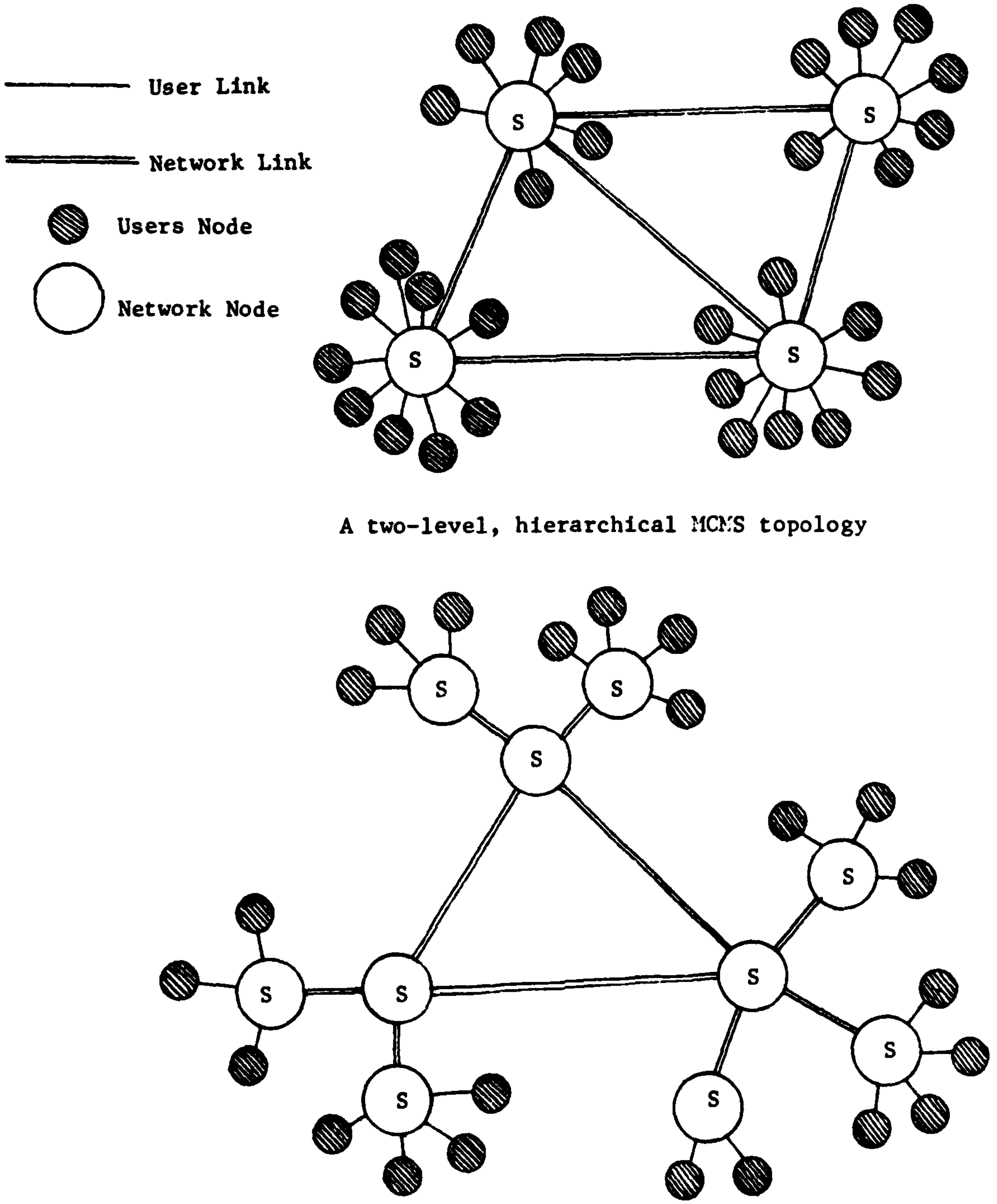
6. single-center,single-star (SCSS) topology,

7. single-center, multidrop (SCMD) topology,

8. multicenter, multistar (MCMS) topology, and

9. multicenter,multidrop (MCMD) topology.

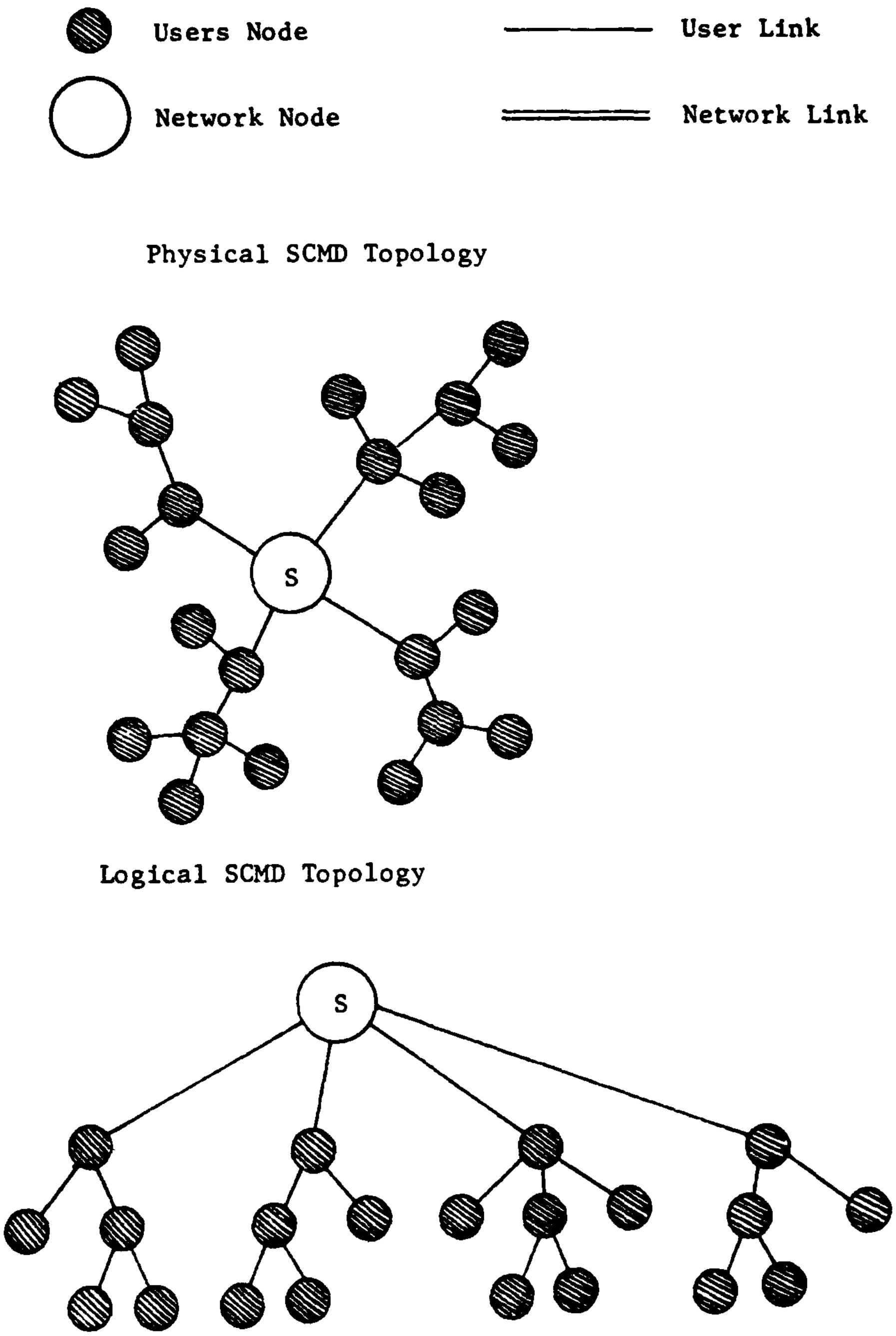
The decreasing cost of hardware favors the implementation of MCMS and MCMD topologies. Two examples of an MCMS are presented in Figure 2-4 while a two-level, hierarchical MCMD is presented in Figure 2-5. Such network topologies require very sophisticated software operating systems. However, such topologies are characteristics of the marketplace in general and not the military in particular. Bus topologies will continue to be popular within the military architecture because they emphasize single processor architectures. The bus structure is a simple and economical interconnection between processing elements and memory modules in a multiprocessor architecture. As a consequence, bus structure are widely chosen during the design of local computer networks based upon distributed control.



A two-level, hierarchical MCMS topology

A three-level MCMS topology

Figure 2-4 Two Examples of a Multicenter, Multistar Topology

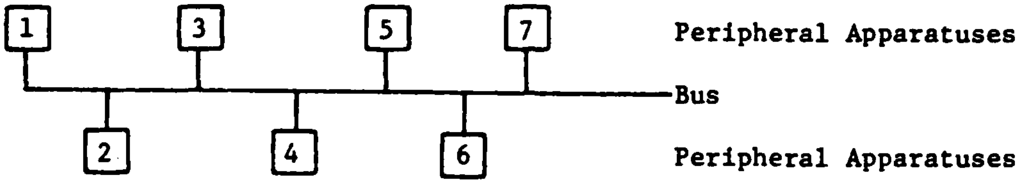


Physical SCMD Topology

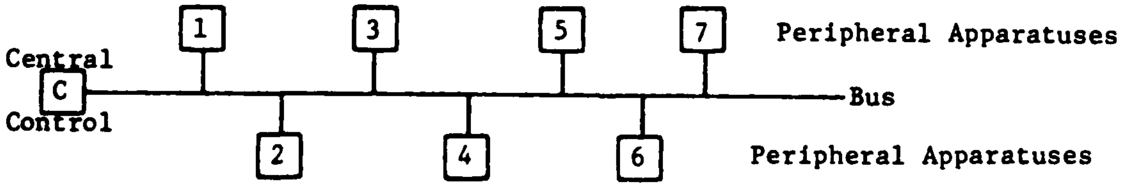
Logical SCMD Topology

Figure 2-5 A Single-Center, Multidrop Topology

A distributed bus topology is presented in the following diagram.



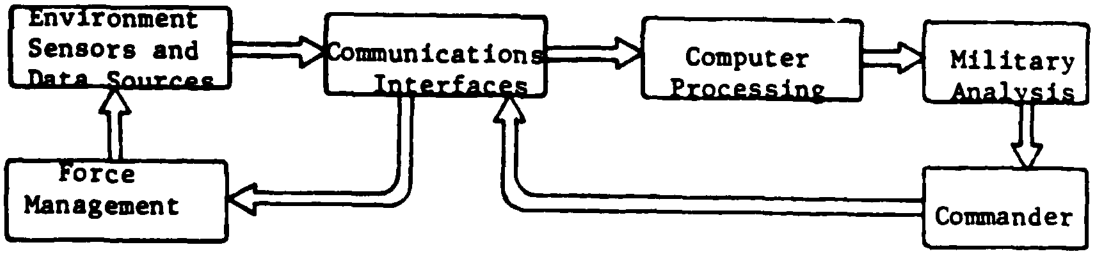
In most instances, the bus itself is controlled by a central controller which is illustrated in the following diagram.



The military situation is typified by advancements within sensors. As more sensors are involved and newer sensors come on-line to existing force management systems, the data rates within existing networks increase. The effect is to shorten decision times. Since the increased data must be accommodated within shorter times, the counter-force capacity must be increased. As this additional capacity becomes available, newer sensors come on-line and the cycle starts again. In such an environment three issues become apparent:

1. What type of system is involved?
2. What kind of distribution is involved?
3. Who is going to use the system?

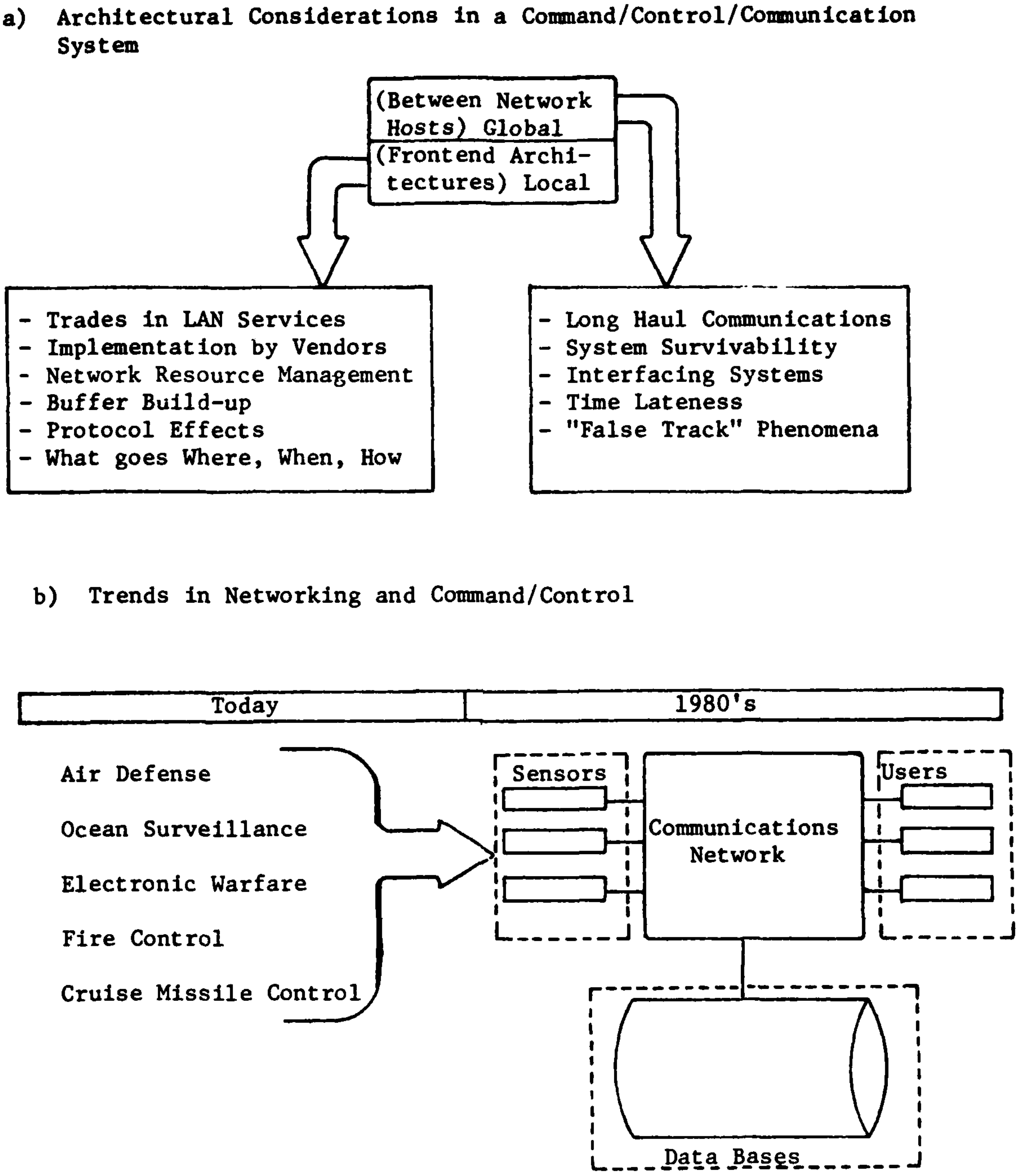
Each issue impacts the definition of software tools. The force management decision making process can be viewed in


Environment Sensors and Data Sources
Communications Interfaces
Computer Processing
Military Analysis
Force Management
Commander

Each situation requiring a decision has two extremes. Furthermore, conflicts between these extremes must be resolved in shorter periods of time. Different decision making scenarios are being studied through simulation, e.g., the Martin Marietta Advanced Modeling System. However, the bottom line returns to network architecture considerations. Figure 2-6 presents such architectural considerations as well as the generic diagram of the military command/control/communication system of the future. The complexity of such a system will be constrained to our ability to verify its software or to simulate its subsequent operation realistically.

2.2 Increased Distribution of Control

Figure 2-7 presents the two levels of control evident in a distributed processing system. In current technology the network operating system resides in only one host and is not replicated throughout all hosts. The Bolt, Beranek and Newman Jericho system is an example. However, replicated copies of the network operating systems present a worst case analysis for consideration. The network level of resource allocation represents the highest level of resource allocation.

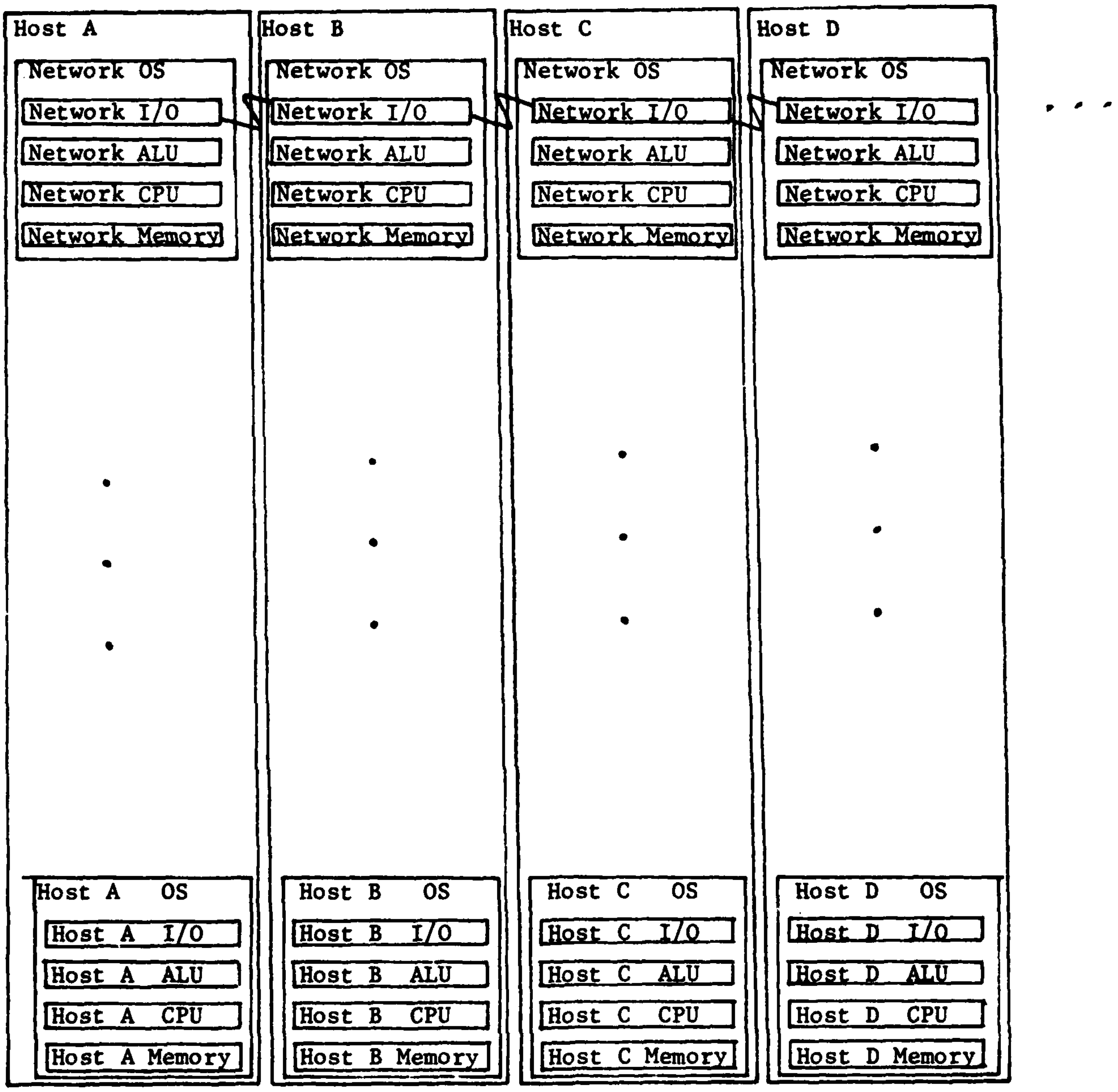


a) Architectural Considerations in a Command/Control/Communication System

(Between Network Hosts) Global
(Frontend Architectures) Local
- Trades in LAN Services
- Implementation by Vendors
- Network Resource Management
- Buffer Build-up
- Protocol Effects
- What goes Where, When, How
- Long Haul Communications
- System Survivability
- Interfacing Systems
- Time Lateness
- "False Track" Phenomena

b) Trends in Networking and Command/Control

Figure 2-6 Architectural Considerations and Trends in Command/Control/Communication Systems



Issues: Should a centralized computer have systemwide executive control to limit the kind of processing which can be embedded in remote I/O systems? Should the network impose little restriction on the physical dispersal of processing and not achieve global executive control?

Figure 2-7 The Two Levels of Control Within a Distributed Processing System

Consequently, work flow is controlled at that level, i.e., jobs are submitted at the network operating system level. In such an architecture the language capability at the network level is a high level one. Consequently, the network operating system itself must not only allocate network resources but also translate its instructions into the operating levels within each network node. To accomplish such objectives is not a trivial task. The leading edges of several issues must be resolved. Object-Oriented Modularization must be applied on two levels simultaneously, i.e., at the network operating system level and at the nodaı operating system level. Conventional Modularization is precluded by the concurrent operating characteristics of the distributed architecture itself. In summary, two great issues arise:

1) Should control be centralized and to what extent? and,

2) What restrictions should be placed upon dispersion?

2.2.1 System-Wide Control With and Without Centralization

Figure 2-7 presented the two levels of distributed processing control. In effect, a single applications environment is presented to the network user. From the standpoint of that user a job is submitted and it is executed. The architecture could just as easily have been a uniprocessor as opposed to several uniprocessors configured into a network (provided execution does not require the concurrent operation of several uniprocessors). The issues faced within the network operating system should remain transparent to the user. To illustrate the complexity being handled by the network operating system, a generic job is submitted on one of the network hosts. Figure 2-8 presents the resultant modularization performed on that job by the network operating system. The job could have been submitted by any node within the network, i.e., Host A, Host B, Host C, Host D, etc. From an executive standpoint the network operating system apportions the job to be performed over the nodes in the network. When more than one node is used the relationship between segments becomes all-important. Since each node operates independently of other nodes, the sequence by which segments are completed determines the validity of results.

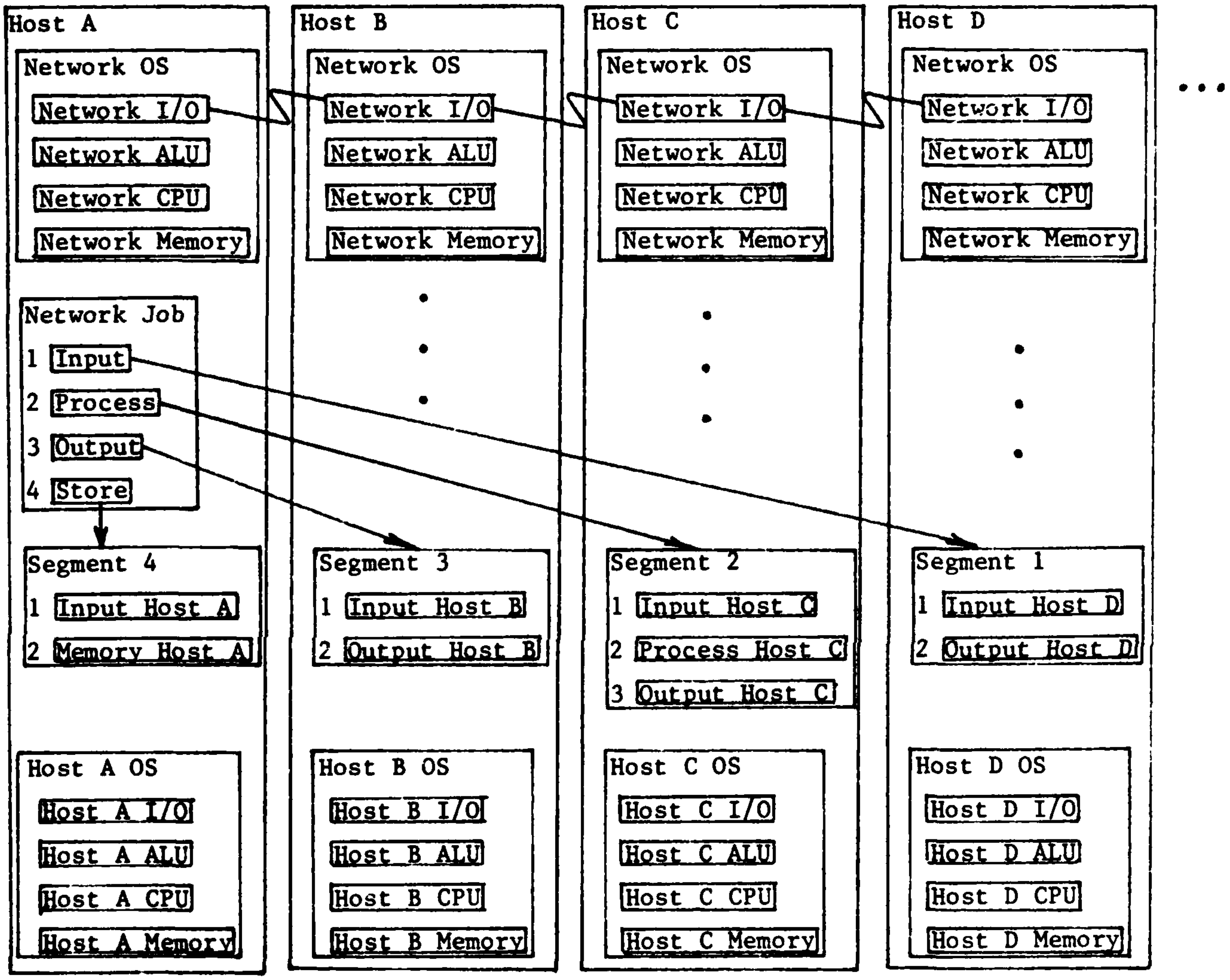
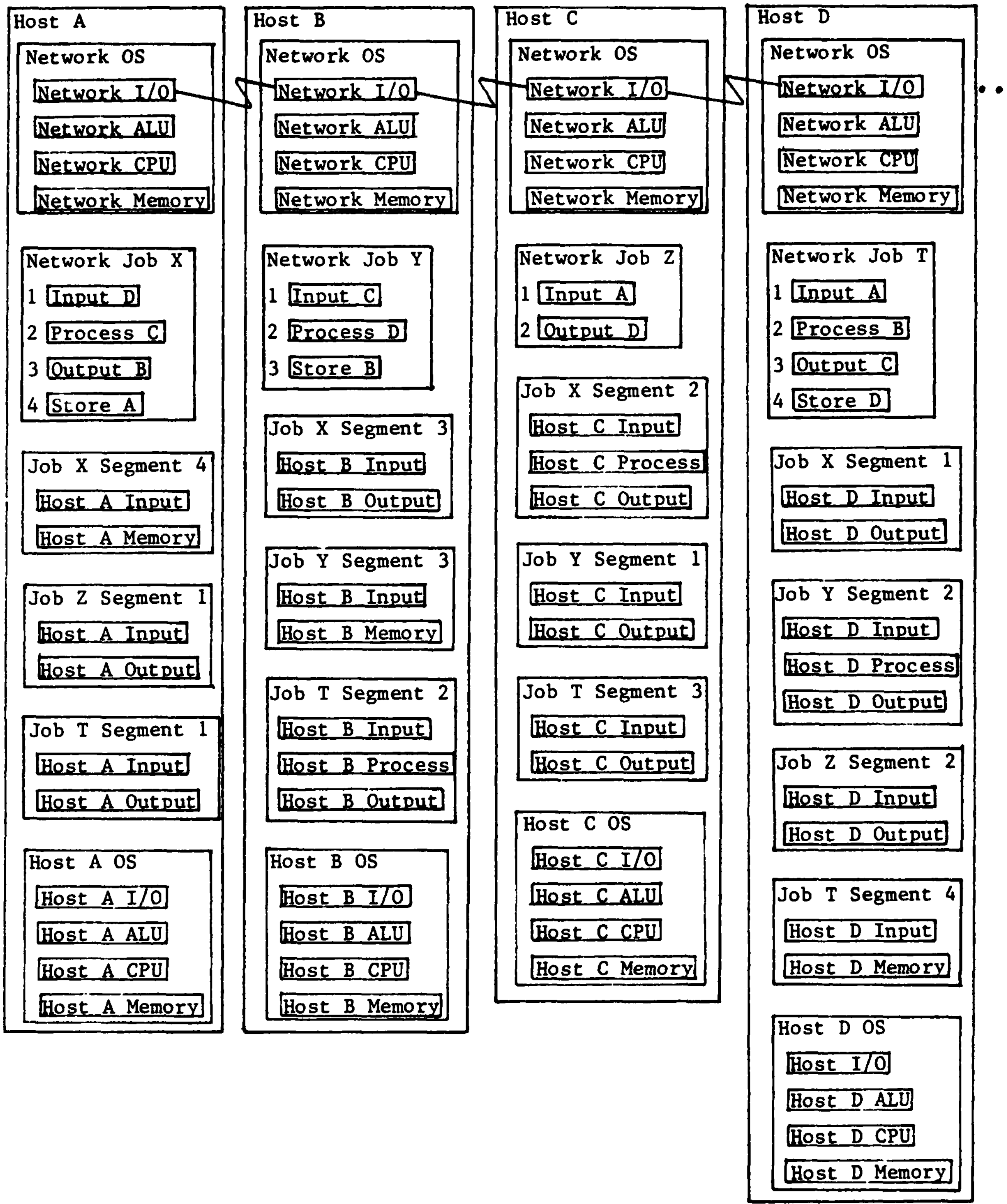


Figure 2-8 Segmentation of a Job Submitted to the Network Operating System

Such sequencing is termed serialization. In the example the assumed serialization is that Segment 1 is completed before Segment 2 which is completed before Segment 3 which is completed before Segment 4. The difficulty enters when many network jobs produce many segments operating concurrently throughout the various nodes of the network. Serialization becomes difficult to maintain. As traffic increases, the serialization problem becomes greater. How the network operating system handles the problem determines subsequent operating characteristics.

Underlying the network is a specific number of nodes at any given point in time. Each node has its own intelligence and probably has its own nodal operating system. In a classic sense these nodal operating systems address the Von Neumann functions of I/O, Processing, Memory, and ALU within each node. Consequently, each node exhibits its own serialization problem. Of course serialization at the nodal level is constrained to sequencing segments exclusively within that particular node. Such sequencing may or may not meet the requirements of the network operating system. How that particular sequencing is satisfied raises the issue of centralized control versus dispersed control. The situation is characterized by Figure 2-9.



Issue: Serialization

Figure 2-9 Multiple Segmentation on Multiple Hosts Within a Network Operating System

Serialization is normally performed within each particular node exclusive of the other nodes. Consequently, ordering from the standpoint of a network requirements is an abnormal situation. In most instances this situation has been controlled by delegating a single host within the network as the network operating system host. In such a context this new host exerts control throughout the network for subsequent queueing of its own host and the others as well. In effect, the network operating system delineates the kind of processing which can be embedded in the remote input/output systems.

2.2.2 Replication to Combat Degradation

Figure 2-7 presented the two levels of distributed processing control. Replicated versions of the network operating system were presented throughout the various hosts within the network. The reason for such replication concerns network degradation. Individual hosts may come up as well as go down without impacting the network. If the architectural philosophy is to allow such coming and going, new problems are created. The resources presented to a network job vary from job to job. Worse yet, the resources may vary within a network job. Such variance impacts the network operating system as well as the individual host operating systems. To accommoaate it is a nontrivial task.

Tightly-coupled resources within a particular local-area-network have the same problem. Accesses to those resources may vary as a local-area-network job stream is processed. The access structure may vary with time, e.g., one processor may go down while its shared memory and another processor continue to function. In fact, the access structure may change during the operation of a single job. To accommodate this coming and going of access structures is also a nontrivial task.

In a concurrently operating environment the resources that stay have knowledge of the operating environment while the resources that go, do not. When such resources return, their knowledge must be updated if they are to assume parity within the operating network. Furthermore, what happened to their network tasks while they were gone? In general, the fewer restrictions imposed on the dispersal of computational power in a network, the more important such questions become. If strong centralized control is not exerted within a network, new kinds of problems arise to accommodate concurrency and degradation.

One alternative to solve the coming and going problem is to build software environments which automatically generate pending tasks for network nodes as they return to service. The same approach seems feasible in software applications for the concurrent environment. In Figure 2-9 the multiple segmentations on various hosts of a network operating system were presented. Using the same tasking, an auxiliary directory can be implemented on each host. That directory can be used by a host to determine the location of tasks allocated for a specific host when that host returns to service. The auxiliary directory is illustrated in Figure 2-10. Assume Host B went down. From jobs accumulated on the remaining hosts, B can assemble its pending tasks from the auxiliary directories when it returns to service. The following table can be constructed.

Auxiliary Directory	Network Job	Instruction
Host A	Job X	3
Host D	Job T	2

Inserting the execution of pending network operating system instructions into a proper sequence is a nontrivial task. Serialization is difficult enough but serialization with network components coming and going is extremely rigorous.

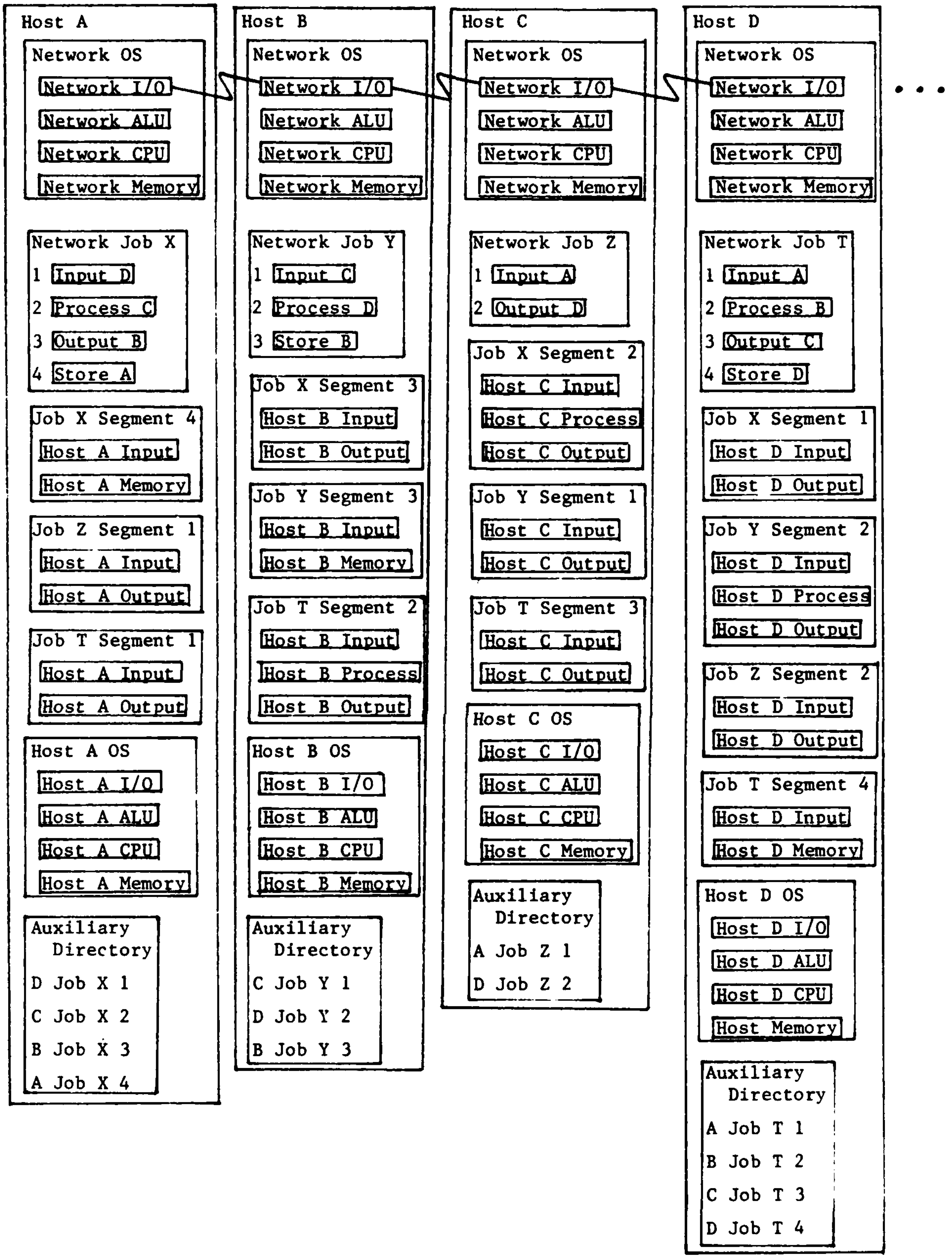
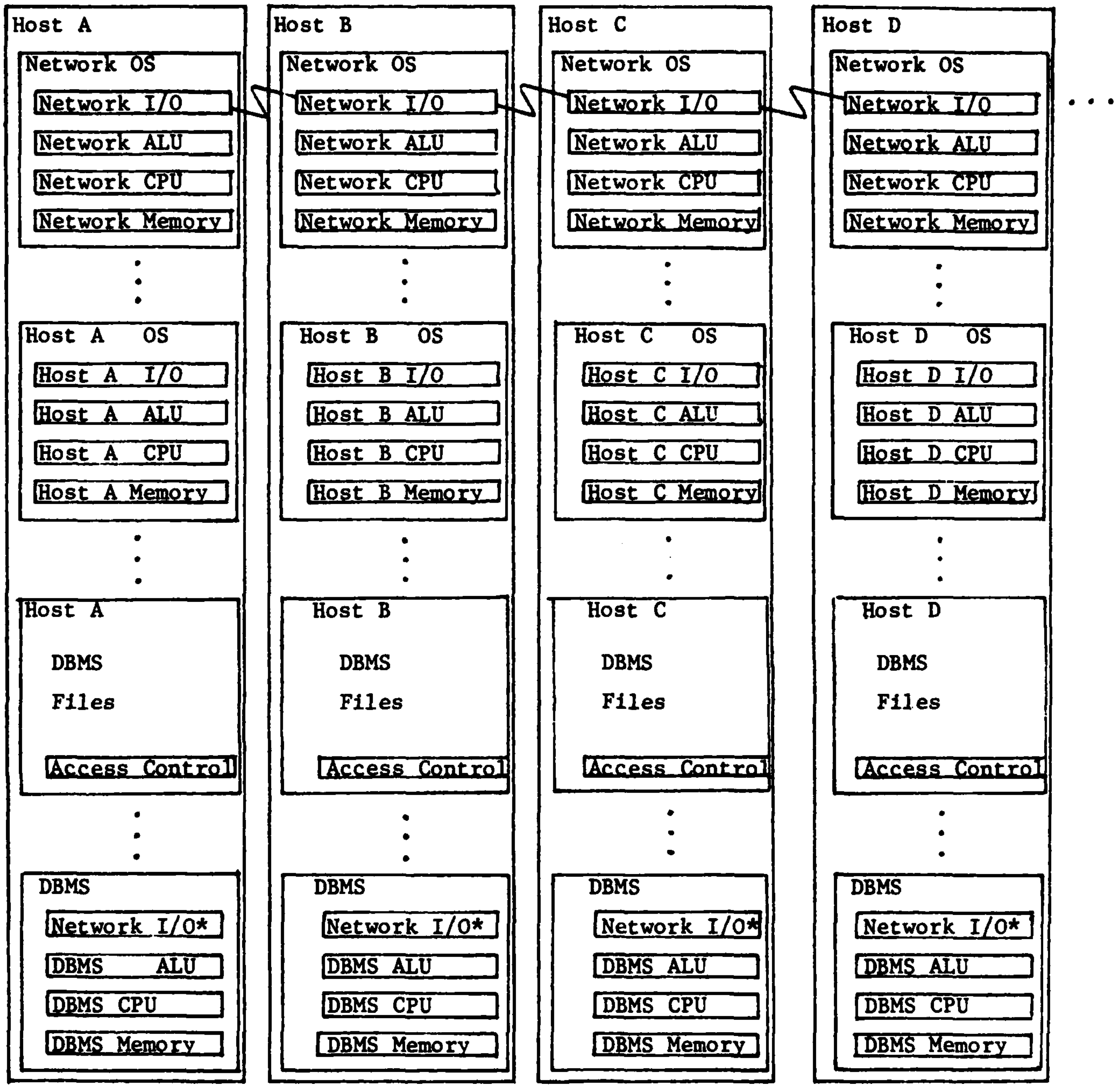


Figure 2-10 Multiple Segmentation on Multiple Hosts Within a Network Operating System Using Auxiliary Directories

However, such coming and going is part of the real world. Consequently, the capability for a network operating system to accommodate degradation and regeneration is an important attribute of a concurrently operating environment. Software designed and implemented for such an environment must rely upon such capability being available. Otherwise the software itself must address degradation and regeneration within its operating environment. Such tools are relatively rare but the need for them is great. Many tradeoffs exist when such problems are addressed. A major consideration is the number of operational restrictions to put into place. Each restriction constrains the computational dispersement within a network. Whether the configuration is loosely-coupled or tightly-coupled matters little. The issues are the same. As each tradeoff is made, it should be carefully documented to enable users to understand its full consequence.

2.3 Increased Distribution of Data Bases

Figure 2-11 presents the four levels of control which impact data bases within a distributed processing system. In current technology the DBMS resides on only one host and is rɔt replicated throughout all hosts.



* Note DMBS I/O depends upon the Network I/O

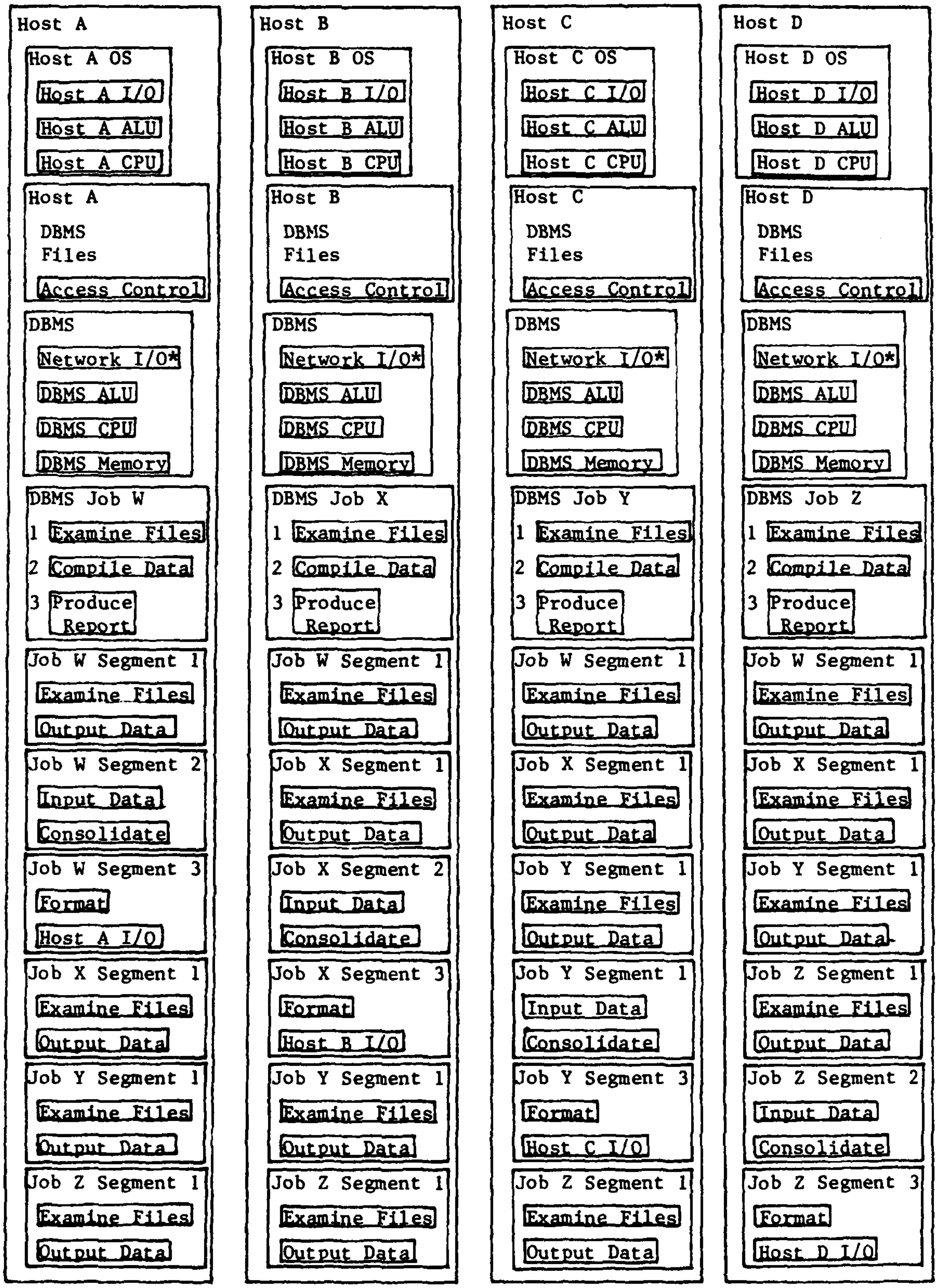
Figure 2-11 The Four Levels of Control
Which Impact Data Bases Within
a Distributed Processing System

However, such replication represents a worst case analysis. Furthermore, as particular nodes may come and go within the network, such an analysis is not too farfetched.

An important relationship exists between distributed data bases and network operating systems. Since such data bases must accommodate input/output throughout the network, they rely upon the network operating systems to accomplish such I/O. Beyond I/O their similarities cease. However, they share several issues in common. In fact, their approaches to such issues must remain compatible with one another. Much work on the compatibilities between network operating systems and distributed data base management systems remains to be done.

2.3.1 Serialization Under Increased Segmentation

The four levels of control which impact the specific host's operating system, the DBMS files residing on each host, and the DBMS itself. These are presented in Figure 2-11. However, that particular presentation does not convey the relationship of the DBMS to serialization and segmentation. To illustrate these concepts Figure 2-12 has been developed.



* Note DBMS I/O depends upon the Network I/O (not pictured).

Figure 2-12 Segmentation of DBMS Jobs Throughout a Distributed Processing System Network

Included in its presentation are the specific host's operating system, the DBMS files residing on each host, and the DBMS itself. The network operating system is excluded since its only purpose is to provide network I/O for the DBMS.

Similarities exist between the operation of a DBMS and the network operating system. Both accept jobs throughout the network. Consequently, the process of modularization/segmentation of these network level jobs remains essentially unchanged. In Figure 2-12 four separate jobs have been submitted to the network at four separate locations. Assuming the DBMS files have been dispersed throughout the network, each job requires searches by each of the separate processors. Once a search has been completed the results must be consolidated over all processors. Once consolidated, the data is formatted and outputed on the originating processor. Obviously the processors need to be carefully orchestrated since they are tightly-coupled from the standpoint of a DBMS. Such an observation is valid even if the actual network operating system architecture is loosely-coupled. Resource sharing within a DBMS can usually be classified as tightly-coupled despite the network operating system. Under such circumstances serialization is not a trivial problem. Serialization from the standpoint of the network operating system can be achieved while serialization within the DBMS goes unaddressed.

To queue the pending segments within individual network hosts from the standpoint of a DBMS requires some network-wide knowledge of the DBMS. Current state of the art exerts such knowledge by applying a single DBMS located on a single processor of a network for control and processing of all DBMS jobs. Such central control obviously constrains the DBMS to a non-distributed, batch-oriented approach. The alternative is obvious but how to establish system-wide DBMS control is not. One key rests with the DBMS files themselves. Access control to those particular files is a means whereby queuing may be accomplished. If the file structures themselves are provided decision-making capability, they can determine when more than one segment is demanding access. They can insure only one segment is operating within them at any given time. Whether this access control interferes with the DBMS serialization from a negative standpoint is a function of how the DBMS is attempting to serialize its segments on a network-wide basis. Both efforts must be carefully orchestrated and designed to operate concurrently.

Since decision-making capability can be supplied the file structures, it can also be supplied the DBMS segments themselves. How these separate intelligences are combined and orchestrated is the subject of much current research. Much remains to be done before such techniques become available in the marketplace. Currently such software resides in the ROMs of the very few hardware products being marketed as database machines. In any case, the distributed architecture is a generation away in computer technology.

2.3.2 Access Control

By its nature a DBMS in a distributed processing environment is tightly-coupled. Its DBMS files are a memory resource shared throughout the nodes of a network. In effect, each node is a processor capable of accessing that DBMS memory regardless of location. Figure 2-12 presented the worst case analysis for a distributed processing network involving four nodes. Each node can accommodate a DBMS job concurrently with every other node. Serialization is a problem summarized in the previous section. However, other problems also exist and are characteristic of distributed networks. Individual hosts may come up as well as go down without impacting the network. If those hosts also control DBMS storage, their presence or absence definitely impacts the DBMS throughout the network. Two areas are of specific concern to the

DBMS viewpoint: file storage and the DBMS modules themselves. In a truly decentralized architecture the DBMS modules are replicated at each node to enable nodes to come and go without impacting each other. However, if those particular nodes also control some aspects of DBMS storage, what becomes of that storage? Obviously if its controlling node goes down, it is no longer available to the network. Consequently, even though the network can continue to operate, a data base request involving its missing file would be severely impacted. The mere presence or absence of a node within a network is not enough for a DBMS to continue to operate. The DBMS storage controlled by those present or absent nodes must be known at all times. The network operating system can gracefully degrade itself while the DBMS may not. Basic issues on a distributed or decentralized DBMS have yet to be resolved. One is the location of tightly-coupled DBMS file storage. As nodes come and go, DBMS files may migrate to stay within an active distributed processing network. Not only is access controlled within such an environment but the files themselves must be capable of migration when nodes go down. Otherwise the DBMS will have limited application to the distributed or decentralized environment.

3.0 Integrated Software Support Environments

Because of their inherent complexity, distributed processing systems are best supported by an integrated software support environment (ISSE). Such an environment provides economy of support through tools which work in conjunction with one another. This eliminates the need for obviously redundant tools which are characteristic of the non-integrated support environments. Integrated support environments also provide tool sets which can be used to address specific problems. The modular nature of such tool sets provides a flexibility which allows problems to be subdivided into object-oriented task statements. Such statements are compatible with emerging languages like Ada, Pascal, and Jovial. Another desirable characteristic of integrated support environments is their open-ended nature. As particular applications need new tools, they can be added. Tools to characterize particular target structures and/or operations can also be added when needed. This capability to add new tools as they are developed makes such environments easy to update and helps prevent their obsolescence.

Open-ended integrated software support environments do have their liabilities as well as their previously mentioned assets. As new tools are added to such environments, extreme care must be exercised. Unless these new tools are added properly, their subsequent use and recall within the integrated environment can be severely curtailed. In such a case, the advantages of the integrated environment will be lost. The probability of such an occurrence is lessened by the inherent simplicity of the tools themselves. Integrated support environments encourage simpler tools since tasks are accommodated by straightforward combinations of less complex tools. As individual tasks are accommodated exclusively by their own tools, tool complexity increases. Although such tools would be difficult to add to an integrated environment, they are precisely the type of tools which are noncharacteristic of it. Consequently, the extreme care which must be exercised when adding new tools to an integrated environment is offset by the simplicity of those tools. What appears to be a liability becomes an asset when the tools remain sufficiently simple. In summary, the utility of an integrated software support environment is a direct result of the simplicity and/or complexity of the tools it contains.

3.1 Impact of Ada

The major impact of Ada is the standardization thrust which accompanies its introduction. As a new language, it also has the impact of any new language which is characterized by the change in features that it provides in comparison to other available languages. Additionally, to meet the DoD objectives connected with the introduction of Ada, the language alone is insufficient and must be supported by a comprehensive integrated software support environment.

The standardization associated with the Ada language is being infused into the required support environment. To facilitate standardization the support environment has been divided into three components: 1) the KAPSE (Kernel Ada Programming Support Environment) which is the host dependent portion of the environment software, 2) MAPSE (Minimal Ada Programming Support Environment) which consists of a minimal comprehensive tool set, and 3) APSE (Ada Programming Support Environment) which is a full environment based upon a particular MAPSE. The KAPSE Interface Team is tasked with standardizing the definition of the KAPSE Interface. Once a standard is established, tools designed to it will be portable to any system with a standard KAPSE. The task of establishing an APSE on a new host will be reduced to constructing a KAPSE for the new host which meets the standard and a code generator for the Ada compiler that is targeted to the new host, along with a rewrite of any target dependent Runtime Support Library routines. This done, the moving of source code of any desired APSE tool to the new host and compiling on that new host is greatly simplified.

Since Ada is a new language it will require the development of the various language dependent tools that are generally available for existing languages. This set of tools is expected to change slightly as Ada was designed to help programmers avoid the known common mistakes. The Ada compilers will be required to provide some checks (previously done by separate tools) in regard to types and range constraints. Ada compilers are also expected to produce set/used listings. A relatively new area for language tools will be the analysis of the concurrency constructs which are available in Ada. Taylor studied the use of the rendezvous mechanism which is used in Ada. His research indicates that it will not be possible to construct efficient algorithms in the general case where no restrictions are placed on the synchronization structure. His work indicates that algorithms can be constructed in certain clases of special situations. The principal feature of these special situations is restrictions in the runtime determination of the scheduling of processes. When the scheduling of processes and process interaction can be made deterministic, then efficient static analysis algorithms can be constructed to detect a wide variety of possible data flow and process scheduling anomalies. Additional research needs to be conducted to ascertain if there are restrictions on nondeterministic scheduling and interaction, which will yield classes for which efficient static analysis algorithms can be constructed.

3.1.1 Host Programming Support

The Host is the system on which a major portion of development and maintenance is carried out. This is the system on which the integrated software support environment resides. Thus to support an APSE a KAPSE must be developed for the host system. Then a MAPSE must be developed which will include tools such as command language interpreter, compiler, linker, loader, symbolic debugger, editor, formatter, database management system, and configuration manager. Additionally an APSE for distributed processing systems will be extended with tools to assist a programmer with handling aspects specific to distributed systems. As Ada matures as a standard, APSE tools will be moved from host to host. The most portable will be generic tools which analyze Ada programs without using implementation or target dependent information. In the near-term many tools exist which are applicable and usable without change. Although these are not written in Ada and therefore cannot reside in an APSE, they can be hosted on the same host as an APSE and be used to augment the capabilities available.

There are also existing tools which can easily be modified to recognize the Ada constructs pertinent to their analysis without a complete rewrite. For the near-term, it will be economical to modify these in the language in which they are presently written. Later, when it becomes necessary to rewrite them in Ada in order to facilitate their installation in an APSE, any desirable modifications which have been discovered during their interim use can be included.

For tools that require significant extensions or rewrites in order to be applicable to Ada, it may be desirable to have them written in Ada. Due consideration must be given, however, to the availability and suitability of Ada environments in which to develop those tools. A cross reference generator for Ada will be required to accommodate the multiple compilation units which Ada supports. In conjunction with this it will need to be able to generate listings by unqualified names and by qualified names. It will need to distinguish overloaded names and indicate which instance is referenced in each case. Resolving overloaded references may be too complex for rudimentary cross-reference tools as some cases are context sensitive and require extensive analysis. In an APSE a compiler can resolve the overloading and store the necessary information such that a cross-reference generator can access it and quickly generate any desired cross-reference listing with any desired level of qualification. Additionally, certain "referenced by" listings will require that a compiler store information in the data base for the modules referenced by the module it is actually compiling. A similar tool, the call tree generator, also has to obtain information which may span several compilation units. This may also be facilitated by the compiler recording pertinent information in the data base.

Static analyzers will be required to process Ada statements in doing many of the now traditional analyses. This will include detection of references to uninitialized variables. It may be desirable that it also be capable of indicating when default initialization is invoked, as Ada has the capability of defining a default initialization for data types. The strong data types in Ada have relegated to the compiler the checking of the legality of data types, the consistency of use of variables, and type matching of parameters. Other data analysis will include the detection of dead definitions of variables. This will need to provide the capability to specify any variables which are memory mapped I/O ports in order that analysis reports will be meaningful.

A relatively new area for static analyzers will be the Ada concurrency constructs. The element of concurrency also adds complexity to previously mentioned static analysis of uninitialized variables and dead definitions of variables. Problems that arise are situations in which a variable is global to two or more concurrent tasks, with referencing occurring in one task and definition occurring in another. The referencing may occur prior to the definition due to a lack of synchronization. Another similar situation is when two tasks may define the common variable, but it may be indeterminate as to which definition will occur first. Again, due to a lack of synchronization, static analyzers will need to be able to analyze the concurrent structures to detect and flag these situations.

Another class of problems is involved with the scheduling and rendezvous of concurrent tasks. Ada may have eliminated the possibility of scheduling a task in parallel with itself; however, it does permit multiple copies of identical tasks, and they may be allocated using an identical name, but the name can only indicate the last task allocated. The Ada synchronization mechanism the rendezvous can permit a number of anomalies. A task may attempt to rendezvous with an unscheduled task, this is not an error in Ada as the task may eventually be scheduled. Static analyzers will be needed to detect when tasks attempt rendezvous with tasks that will never be scheduled. A task may also attempt to rendezvous with a terminated task or a task which terminates prior to servicing the call. This will generate a runtime exception in Ada. It will, therefore, be desirable to have a static analyzer to indicate where and under what circumstances this situation can occur. It is also possible that a task enter a state in which it will never service certain of its entries. Other tasks which attempt to call those entries will wait forever. Situations of this sort need to be detected by a static analyzer. Ada has a restriction that a block, subprogram body, or task body may not be left until all dependent tasks have terminated. This may lead to situations in which a task is deadlocked because a dependent task is in a nonterminating state or cannot proceed to termination because it is waiting on an event which will not occur. Detecting these situations with a static analyzer is desirable. As previously mentioned, this will be difficult for programs in which the scheduling of tasks is nondeterministic.

A similar class of problems deals with the allocation and termination of tasks. In Ada, tasks are not necessarily dependent on the block, subprogram body, or task body in which they are allocated. It is, therefore, possible to allocate a task using a local variable, then exit and lose any means of accessing that task. This is not always an error, as the intention may be to start up an independent active task with which no additional interaction will be required. This may even occur at the main program level, as a task may be dependent on a library package, and therefore, may not be required to terminate prior to the completion of the main program. Such an occurrence generates an operational task which appears to be an orphan, i.e., it has no living parents. A static analyzer should easily be able to discover the tasks which belong in the above mentioned categories and provide a list of them for consideration. Another scheduling problem can occur when an unlimited number of tasks can be generated without requiring that any terminate. These situations are difficult to dissect with a static analyzer as they are quite often very dependent on external stimulus. A static analyzer should, however, be able to flag these areas for further examination. A thorough static analysis of task scheduling would produce statistics on how many of each type of task could be in each queue. An analysis this thorough is ambitious even for deterministic scheduling cases and likely impractical for many nondeterministic schedules. The problem of orphan tasks exacerbates the situation.

3.1.2 Target Programming Support

Ada's main impact on the Target Programming Support is that it is a language that supports concurrent aspects in programs. There are several features that current research has recognized as required to support distributed processing. They are: 1) a basic software unit for distribution, 2) a means of exchange of information between units, 3) a means of synchronization between units, 4) a control structure to handle nondeterminism, and 5) a kernel to interface between high level program language and hardware. Ada provides its particular brand of each of these features.

One is a basic software unit for distribution which is embodied by the task in Ada. They may be specified at compile time or allocated dynamically at runtime. The Ada loop construct permits a run forever version of a task. However, a task may terminate by completing its code or by a terminate statement in a selective wait statement. It can also be the object of an abort statement. The maximum permitted number of active tasks is limited only by available resources. There is some control over resource utilization provided to the programmer through the specification of storage space allotment for a task or task type. The interrelationship between tasks is hierarchical as each task is dependent on the block, subprogram body, task body, or library package in which it or its access type is declared. Calling another task, however, is limited only by visibility rules. Thus a programmer has a great deal of discretion in the call structure he utilizes. The call mechanism known as the rendezvous in Ada is a well defined synchronized interaction mechanism. Another means of interaction would be via global variables for which there is no implicit control other than normal scoping rules.

A second necessary feature is a means of exchanging information between tasks. This is supported by the above mentioned rendezvous in Ada. This high order language construct hides the hardware configuration from the program level software. It utilizes the very powerful and general technique of message passing. Automatic buffering is not provided; therefore, the first task ready to communicate is blocked until the other task is ready.

Because of this blocking, the rendezvous also satisfies the third required feature which is a means of synchronization. In Ada the rendezvous is not required to include a parameter list for message passing, thus parameter passing overhead is not imposed on rendezvous used simply for synchronization. An additional feature included with the Ada rendezvous is a critical region of code which is guaranteed to be executed prior to the calling task being released to proceed with its own execution.

The fourth feature required is a control structure to accomodate nondeterminism. This is provided by the selective wait construct in Ada. There are additional select constructs which provide for conditional and timed delay on rendezvous requests. Conditions may also be associated with each possible rendezvous in the selective wait construct.

The fifth feature is a kernel to act as an interface between high level program language and the hardware. A kernel, because of its interface role, is extremely sensitive to the hardware characteristics as well as the language. The Ada language definition does not address the kernel. Ada is intended to be used on a variety of target systems; therefore, a specific target system hardware has not been defined. There is a move to define specific kernels by developing formal requirement specifications for Ada Target Machine Operating Systems for the target machines used in military systems.

In the distributed processing environment, the target operating systems must provide not only an interface to the hardware but must also support an interface to the distributed structure of the entire target system environment. It must correlate the Ada tasks, which are software units for distribution, to the distributed processing units in the hardware system. This a non-trivial problem and has a multitude of possible solutions. Some possibilities are one Ada task per processor, or any number of Ada tasks running indiscriminately or any number of identical processors, or selected groups of Ada tasks running on specific different processors. There are other more complex possibilities such as systems which permit Ada tasks to migrate from processor to processor via subprogram calls. Particular associations between tasks and processors supported by individual target systems will vary greatly. For this reason system designers will need modeling tools to support rapid prototyping and simulations in order to try out various possibilities and make intelligent decisions concerning the best target system for each specific application.

Scheduling of tasks is another aspect which must be supported by the target operating system. This can vary greatly both with what is supportable by the particular target system and with the requirements of the particular application program. In some systems scheduling may be deterministic, in others nondeterministic. In conjunction with scheduling, allocation of resources to tasks may be static or dynamic. Complications arise when tasks in one process can initiate or invoke the scheduling of tasks in another processor. In some systems the control of resources will reside solely within the network operating system. In other systems there will be a need for Ada implementation pragmas which will provide limited control of resources to the application program level. Again, modeling of resource allocation will need to be supported so that various schemes may be evaluated by the designers prior to commitment to a particular scheme for implementation.

Another level of support in the operating system is for the intertask communication embodied by the rendezvous in Ada. The operating system must provide an interface between this software communication mechanism and the actual hardware communication between distributed processors. The solutions available here are closely tied to how tasks have been distributed throughout the system.

If tasks have been assigned one for one to processors then the software rendezvous can be implemented directly by the hardware communication mechanism. In systems using a homogenous structure of identical processors the rendezvous could be supported strictly at a software level, thus only indirectly affecting hardware processors through its affect on task queue status. For all the other various system structures the rendezvous support may require more customizing in order to accommodate rendezvous between tasks residing in the same processor or processor group, and to accommocate rendezvous between tasks in separate processors either identical or of diverse types. Passing rendezvous information between processors is complicated by the blocking nature of the rendezvous since in most cases it is desirable that only the task and not the processor be blocked. These types of interactions will not only need to be prototyped during early states of design, but also need to be exercised in a full scale simulation or on the actual target system during the coding and implementation phase in order to tune the software to provide the desired response.

3.2 Design and Development Considerations

With the near-term certainty of distributed computing systems, as both host and target, much attention must be given to the methods and vehicles used for system development. The development of distributed systems is in some ways similar to the development of conventional, centralized systems, but in many ways far different. Workshops must be aware of the fact that environments that support centralized systems development cannot simply be "massaged" slightly to accommodate distributed systems development. Rather, an ISSE must be built for the specific types of distributed systems to be developed.

As discussed in section 2.1, the current military systems range from centralized to what can be termed "moderately distributed" (see Figure 2-2). For the near-term then, tools needed to build very loosely coupled systems need not be included in a military ISSE.

The rest of this section deals with the types of tools and methodologies that are of prime importance for building an ISSE for military use. First, some overall policies and basic tool requirements are presented. Then, the specific impact of distributed operating systems, interconnection architectures, and data bases is presented.

Design methodologies and the methods by which these methodologies are conceived must be altered to reflect the nature of distributed systems development and the problems inherent to it. The main responsibility for prevention, detection, and correction of errors must be assumed by the requirements (taken here to mean both requirements specification and analysis) and design phases of the software life cycle (see Figure 3-1). Conversely, the coding, testing and maintenance phases must be relieved of as much of the responsibility for system soundness as possible. The main reasons for this are:

1) EASE OF CORRECTION & DETECTION - If errors are detected and corrected in the requirements and design phases, much less effort is required to correct these errors than after they are "hard-coded" in the implementation phase.

2) COMPLIANT SOFTWARE - Traditional error correction and detection (coding, testing, maintenance phases) leads directly to noncompliant software which can radically shorten the system's life.

A more detailed discussion of these issues follows.

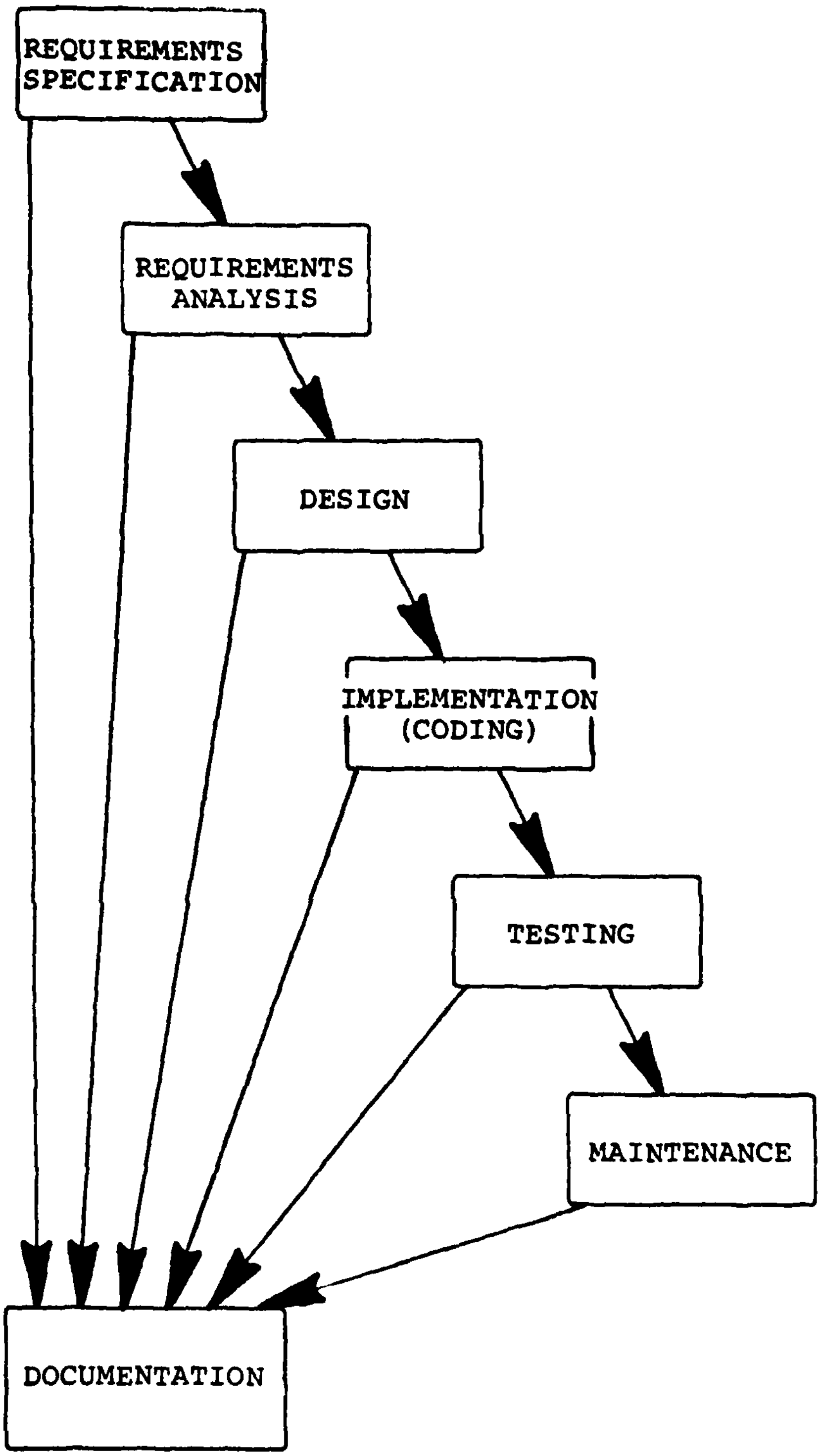


Figure 3-1 Generic Life Cycle Phases

Since the definition of distributed systems states the presence of more than one node, once the software is coded it becomes truly distributed. Software errors usually impact other elements of software (and usually other nodes), and therefore, error detection and correction requires the identification and correction of any and all software impacted by that error. For example, if it is decided that a data type be changed, then a maintenance programmer must find all statements and declarations that reference that type and update them accordingly. This task is extremely time consuming, and system degradation is almost certainly accelerated. When the software is distributed over many nodes, totally repairing an error is a very difficult task for a programmer to perform without the aid of tools, and may leave the system in worse shape than before. If the error is detected in the requirements or design phase, correction is far easier and system integrity is maintained much longer.

The insurance of software compliance, or implementation that complies with its design, is another argument for shifting error detection and correction "upward" in the software life cycle. If errors are detected and corrected after the final design, it is easy for the software to become noncompliant and, therefore, not as maintainable as it should be.

This upward shift of responsibility in the software life cycle must be implemented by development methodologies and the framework used to form these methodologies. This framework must be designed to produce methodologies that place special emphasis on the requirements specification, analysis, and design phase of the software life cycle.

With the importance of design methodologies recognized, we can proceed to a discussion of the creation of an ISSE that sufficiently supports production of distributed systems. This ISSE will be composed of a standardized minimal tool set (see section 3.1), as well as all the tools necessary to accommodate any and all methodologies that might be developed. The remainder of this section discusses the major tool needs and sketches outlines of their design.

There exist several areas of tool classifications that need improvements or extensions to make them useful to the software engineer who is building a distributed system. Many of the tools that exist now and were designed for use on centralized systems development lend themselves well to distributed systems as well. It is not that old tools will no longer be useful in the distributed environment, but rather that more tool support will be needed due to the non-deterministic nature of concurrent software. The areas of particular need are:

1) AUTOMATED SPECIFICATION LANGUAGE/ANALYSIS - Network communications are not specifically addressed by any current specification language, and this area should be the one most stressed for tool development.

2) STATIC/DYNAMIC ANALYSIS - The existing tools need to be extended to tell programmers when the possibilities exist for certain concurrent software phenomenon.

3) SOFTWARE INTERRELATIONSHIPS - The internodal dependencies of all software in a distributed system need to be permanently catalogued to reduce time and cost related to the testing maintenance phases.

A more detailed discussion of these areas follows.

The methods currently used to assist in the systems requirements and design phases are insufficient to fully support development of distributed systems. Since it is of paramount importance to give the development of distributed systems maximum support at the requirements and design phases, new methodologies and tools must be developed to further automate these phases. Automation of these phases is the key to making the design methodologies as useful as they should be. Specifically, they must address the issues of protocol definition and bulk data communications, to assist in determining optimum networking methods. These tools must also produce reports that are easily reviewed and modified by humans and then fed back to the computer to be reanalyzed. So-called "Feedback Development" must be used when developing distributed systems.

Once the requirements and design phases have been completed, the programmers charged with implementation of the defined system must be provided with tools to aid them in producing sound distributed software. If the methodology being used is a good one, the programmer will be provided with a specific design with all of the distributed processing considerations already addressed and resolved.

The problems arise when program errors inherent to distributed processing occur and no testing tools exist to detect or prevent them. For example, orphan spawning (see section 3.1.1) and deadlock are two of the problems that arise in distributed software and further study is required to specifically address these problems. Static analysis techniques such as path analysis can be extended to tell programmers when the possibility for these phenomena exists, and the programmer can then investigate further. For example, if a programmer wants to test his code for orphans, static test tools (extensions of existing ones, that is) can analyze the program and identify points in the code where orphan processes might be spawned and which processes they might be. Because of the non-deterministic nature of distributed software, current static analysis tools can do no more than this. Dynamic analysis tools also need to be extended to allow for detection of orphan processes via instrumentation schemes.

Finally, the problem of software interdependency must be addressed. Distributed software that is non-deterministic operates as separate autonomous entities, and maintenance is extremely difficult. When one area or module of software must be altered, the impact on other modules is usually far reaching and unpredictable. Since maintenance represents approximately 75% of the software life cycle, and even more in the distributed environment, tools and methods must be incorporated into any ISSE to help categorize software interdependencies. Although tools that manage these types of software interdependencies exist, there are several shortcomings with them. First of all, the data is not managed by computer, and usually takes the form of a post-mortem listing. Therefore, no categorization is performed and no easy cross-reference ability exists. Second, the only way to build an accurate final copy of the relationships is to manually update when modules are recompiled. In a large distributed system, this task represents quite a problem. Therefore, a tool must be developed to completely automate these data collection and management functions. Figure 3-2 depicts a tool that makes use of the symbol table built by the compiler to collect software dependency data. This tool builds and maintains a permanent data base consisting of this symbol table information. This data base can then be queried interactively by programmers or evaluated by the static analysis tools of the ISSE. Programmers could then determine all of the changes required to repair a problem and avoid hasty and ill-advised "patches".

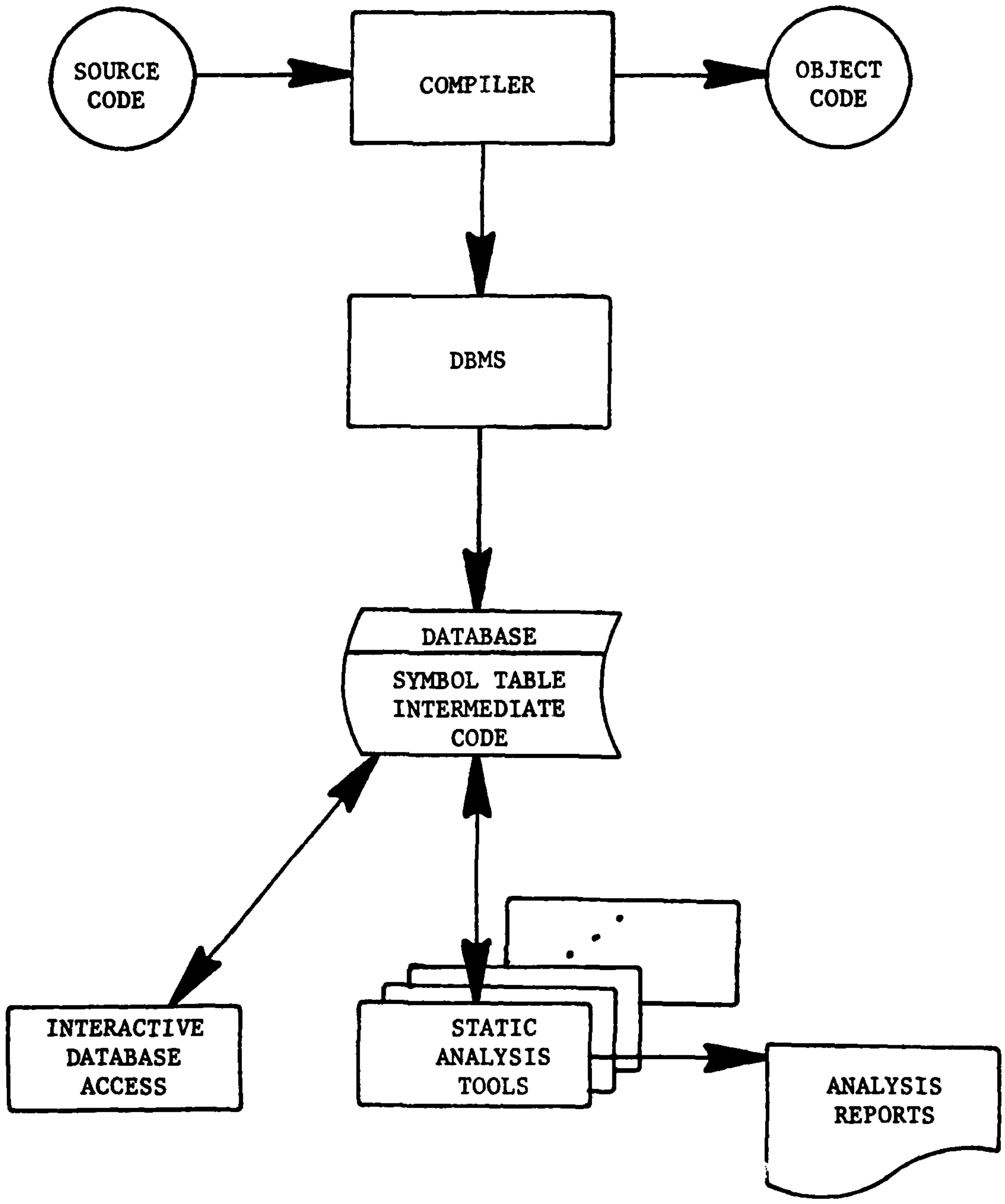


Figure 3-2 Data Flow Diagram of Generic Software Dependency Tool

This technique would greatly slow the phenomenon of "software rot", a major problem with distributed software.

These tools designed and added to a standardized tool set will allow for more efficient and complete distributed system design and development. Most importantly, the methodologies adapted by a particular workshop should all be totally supportable by any such ISSE.

Any ISSE built to support distributed systems design and development should have the characteristics outlined by the preceeding section. Also, the specific types of distributed systems to be created has an impact on which tools comprise the ISSE. Following is a discussion of the three most important variables used in describing distributed systems. The military's near-term target computer systems are analyzed with respect to operating systems, interconnect architectures, and data bases, and the impact of each of these on the ISSE is presented.

3.2.1 Distributed Operating Systems

Distributed Operating Systems (DOS) are the entities that coordinate the activities of many concurrently functioning processors and other resources. The scope of this section is limited to a discussion of DOSs only and not individual

Constituent Operating Systems (COS). Distributed processing considerations only impact COSs when it is being determined how much, if any, of the COS responsibilities will be relegated to the DOS when designing a network. This section also limits its discussion to the military's current and near-term distributed systems technology, e.g., low to mid-range distribution of control (see section 2.1). The basic functions of a DOS and its design considerations are discussed with respect to the military's two basic distributed processing areas: communications systems and weapons systems.

Although communications systems and weapons systems occupy mutually exclusive volumes in the distributed processing three-space pictured in Figure 2-2, the design and development considerations of the DOSs for these systems are very similar. The functions which both types of systems must provide are the same, though these functions vary in relative importance. Three basic functions are:

1) Resource management (including data transfer/communications)

2) Fault tolerance/recovery

3) Transparency of system control.

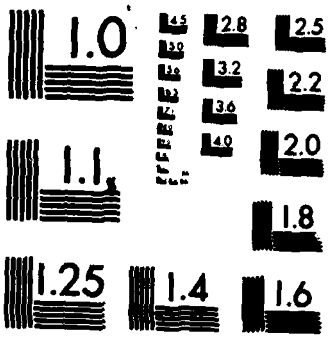
Resource management is the main function of executive control (DOS). This is the function of sending messages and coordination of the different nodes of the network. These resources consist of all the separate entities of the network to be united into a single functioning whole. Communications systems executive control typically manages a large quantity of data transfer devices, e.g., satellite communication links, packet radio controllers, as well as the standard types of nodes. It is important for most communications systems to be easily reconfigured, relocated or added to quickly, so the executive control must lend itself to this dynamic resource configuration. Weapons systems, by contrast, are more static in their configuration but their resources demand a high level of coordination by the DOS. This is because of the stringent real-time environments in which they operate. For instance, up to date information on the state of all processor queues must be kept or quickly obtainable to insure the high throughput of time-critical tasks. So, though in each environment (communications, weapons) the DOS must place emphasis on different resource management issues, the same basic functions are performed by each type of DOS.

Fault tolerance and recovery is another main function of the DOS and, like resource management, receives different emphasis depending on the type of system. Extreme fault tolerance is usually required of weapons systems and the DOS must be designed to accommodate this. High replication of hardware and a highly distributed DOS, usually with multiple autonomous points of control, insure this high level of fault tolerance. In the case of communications systems, where faults can be tolerated relatively more often, more emphasis is placed on quick, state-resuming recovery. In these systems, though hardware is often replicated, the DOS is uaually of a dynamic master-slave nature. This usually takes the form of one processor possessing all the DOS modules and functioning as master, but upon an abend, one or more "slave" processors are capable of assuming possession of the DOS modules and becoming master. In this environment, the DOS is charged with the responsibility of maintaining a high degree of state information in its local tables for efficient recovery purposes. Also the various network nodes (especially communications processors) are designed to retain recent data transmissions for a short time in case the current master abends and another processor must assume master status. In this way, fault recovery can occur with a minimum of state information lost.

The network operating system must also provide transparency of system control (a virtual machine layer) to all

AD-A137 687 DISTRIBUTED PROCESSING TOOLS DEFINITION VOLUME 2 2/2
APPLICATION OF SOFTWARE..(U) GENERAL DYNAMICS FORT
WORTH TX DATA SYSTEMS DIV H C CONN ET AL. JUN 83
UNCLASSIFIED RADC-TR-83-107-VOL-2 F30602-81-C-0142 F/G 9/2 NL

END
FILMED
3-84
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

user and most applications programs. The exception to this is found in some embedded weapons systems, when applications must be programmed using knowledge of the network configuration. In these real-time systems, the DOS functions mostly as a communications supervisor, turning much of its responsibility over to the application software. But usually the DOS must provide a level of transparency allowing user and application to proceed without specific information about the different network nodes. Reference by name, rather than address, is an example of this. In communications systems, this DOS function enables packages of messages to be sent by users to people without concerning themselves with the specific location and routing information. Flexible DOS protocol techniques implement these communications and allow for easy reconfiguration of the network. Likewise, the DOS in most weapons systems allows for applications software to be designed and run as processor-independent code. Also, interprocessor communications are handled by the DOS allowing, for instance, an application program to reference data without knowing which of many data bases it is stored on. Such flexibility serves to greatly enhance fault tolerance and retard system degradation.

The design and development of these distributed operating systems is quite obviously complex and the overall performance of the resultant network depends highly on the soundness of this design and development effort. As is the case with distributed target software development, DOS design and development must be accomplished within an ISSE. The extensions to be made to the ISSE discussed in sections 3.0 and 3.2 in order to accommodate DOS development are few. The need for tools and methodologies in the requirements specification phase, the requirements analysis phase, and the design phase, is particularly pressing when developing DOS software. Specifically, simulation tools that expose the operating characteristics of models of proposed DOSs must be incorporated into the ISSE. Several simulation tools exist today that test the operating characteristics of most of the network models being proposed. The only shortcoming is in simulating interfacing between nodes that are not compatible as far as protocols and data structures. The encoding, decoding, and transmission of data between nodes using non-standardized protocols needs to be simulated accurately in order to satisfactorily evaluate proposed models in the design phase.

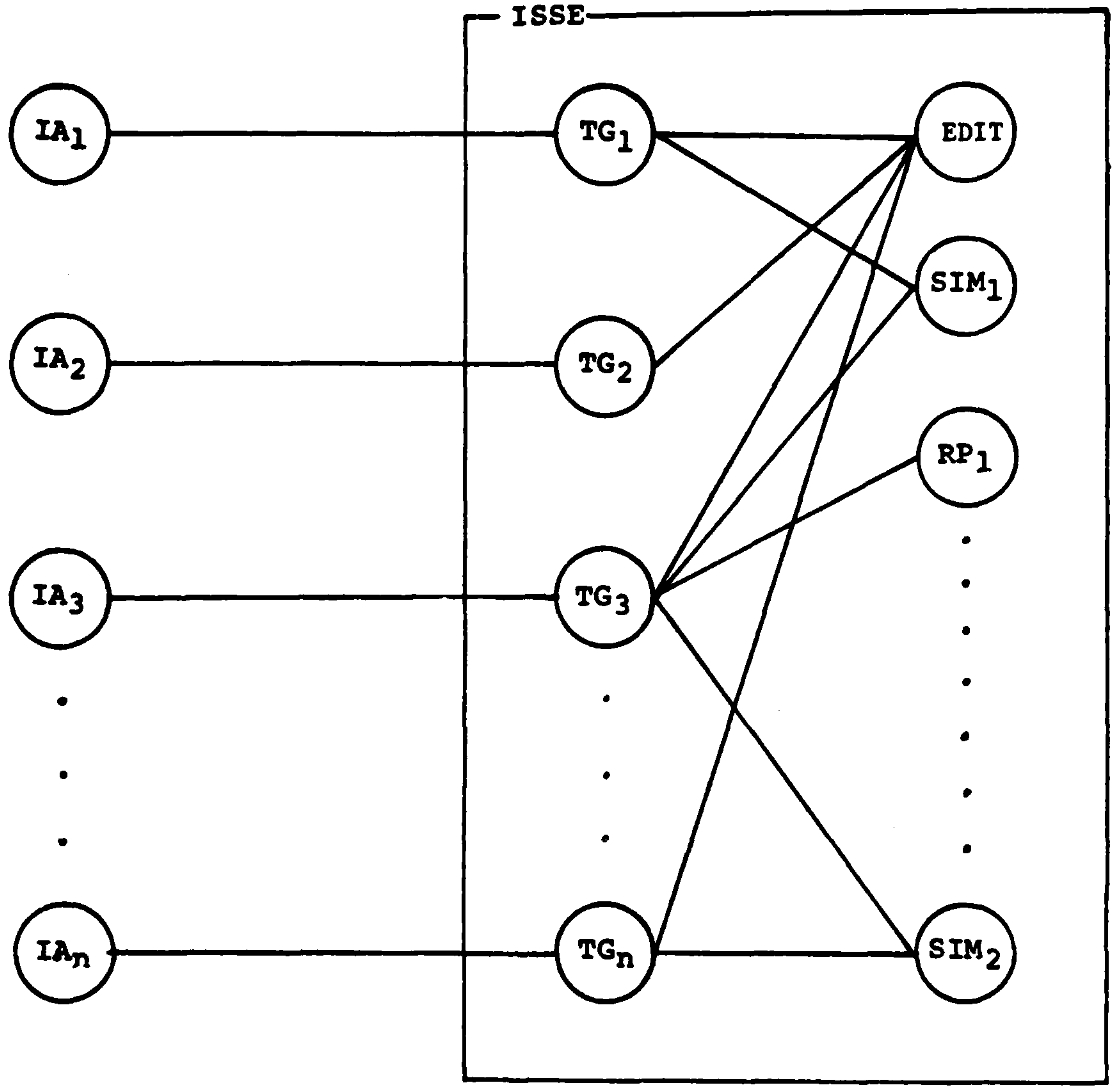
Beyond the need for this added simulation capability, the ISSE discussed in earlier sections will prove to be sufficient to support the design of distributed operating systems. When development of the DOS is conducted separately from the development of applications software targeted for the same system, i.e., most weapons systems, the same methodologies and ISSE will be sufficient to provide sound support to both efforts.

3.2.2 Interconnect Architectures

The means in which various components of a network are interconnected has a direct and profound impact on overall system performance. The schemes for interconnection, or interconnect architectures (IA), can vary greatly from system to system. Therefore, system designers must be provided with the means to fully evaluate the advantages and disadvantages of various proposed architectures before committing manhours of labor to implementation and detailed design. This function must be provided by the ISSE (see section 3.0). This section discusses various ways of classifying IAs and the types of tool support needed within an ISSE for modeling, prototyping and simulating these IAs. Note that specific architectures and design criteria are not addressed in this section. Rather, software design and support considerations are the primary concerns. Finally, this section discusses some second-level issues of the interconnection of various networks and the problems in designing tools to support these types of architectures.

Several major researchers have made attempts to categorize different types of IAs. The one common factor between these attempts is a range in IA design from very loosely-coupled architectures to very tightly-coupled ones. The various ways this continuum is broken into distinct classifications is not important to this discussion. What is important is the fact that there exist different classifications of IAs, and tool support must be grouped in the same classifications.

Figure 3-3 illustrates a proposed grouping of support tools. For each IA category, IA , a corresponding group of tools, TG , must be provided in the ISSE.



LEGEND

lA_i = INTERCONNECT ARCHITECTURE
TG_i = TOOL GROUPING
ISSE = INTEGRATED SOFTWARE
SUPPORT ENVIRONMENT
SIM_i = SIMULATOR
RP_i = RAPID PROTOTYPER

Figure 3-3 Tool Support for Interconnection Architecture Classifications

Different tool groups may consist of many of the same tools. For example, the analysis and design of IA can be completely supported by TG . This group may consist of a text editor, one or more simulators, a rapid prototyping tool, etc. Some or all of these tools may also be part of another tool group, but the union of all the tool groups is the total set of tools needed to support any IA design.

As indicated above, the requirements phase of the software life cycle is where most of the tools to support IA design are required. Specifically, rapid prototyping, simulation and modeling techniques and methodologies must be refined/developed to support the entire spectrum of IA possibilities. There are several aspects of distributed architectures that suggest changes required to simulation tools. The remainder of this section deals with these issues.

The current general method for iterative modeling is top-down. That is, a model of the overall system is developed, then simulated, and from the results of the simulation a new, more specific model is developed. This process continues until the results are sufficient to make sound design decisions or until the simulator tool(s) being used can get no more specific. Unfortunately, the latter is usually the case. For this reason, and because networks are becoming increasingly complex and layered (e.g., networks within networks), it will be necessary to use simulation tools to simulate various nodes of a distributed system while the overall system is itself being simulated by a tool using the data from the nodal simulators. Take, for example, the design of a distributed system consisting of four major nodes, A, B, C, and D. Each of these nodes is in turn a network of its own and may or may not be implemented already. It is extremely advantageous to be able to test different IAs before deciding on a final configuration for the overall network. To this end, the four nodes A, B, C, and D can be simulated while another tool concurrently simulates the operation of the entire system using the simulation of A, B, C, and D as inputs. As stated earlier, the tools picked for incorporation into the ISSE must be able to support the design of the entire spectrum of IAs. Also, they must be capable of operating concurrently together as any level of a proposed network. Much further study needs to be done in this field before tools that possess this functionality can be built.

Keeping the ultimate goal of a highly functional, modular set of simulators in mind, a repertoire of simulators can be built up in ISSEs. There already exist network simulators which are quite flexible and powerful. They need to be extended to reach the desired goal of functionality. Such extensions are necessary within the near-term time frame.

3.2.3 Data Bases

In section 2.3.1 a discussion of data bases in the context of a decentralized system was presented. However, such a discussion is incomplete until the impact of data bases within the target environments is discussed. These target environments are the weapon systems problems. State of the art development systems involving host-to-target downloading characteristically operate on a point-to-point basis. In other words, a single target's program is developed and downloaded from the host environment. When programs for more than one target are involved, they are developed on a sequential basis for one target after another. Underlying such approaches is an implicit assumption that targets themselves are controlling their own resources, e.g., memory and mass storage. In reality such control is often shared, e.g., two targets accessing the same memory buffer. This happens in a tightly-coupled architecture and when it occurs the data base can be described as the contents of that shared memory buffer. The complicating factor is that such a data base is not under the complete control of either target. Furthermore, when one target goes down, the other continues to operate as long as the memory buffer functions. If that memory buffer malfunctions, both targets lose access to the data base. Of course the data base could be important enough to be provided redundant storage, i.e., an alternative memory buffer in which to reside. In either instance, the applications development environment of a multiple target configuration is not a straightforward use of point-to-point communications between host and target.

Application programs residing within targets manage the databases controlled by those targets. Such management addresses the following issues:

- how the databases are structured;
- how interconnects between targets are accomplished;
- how the "typical" application is accomplished;
- how the data model is structured;
- how the targets are synchronized;
- etc.

The key motivation behind such environments is high data base availability and accessibility. When data bases are shared between targets, the objective is to increase accessibility while enhanching reliability. In fact, the reliability of a system composed of several targets is greater than the reliability of each single target. However, such gains in reliability extract a price in terms of data base software. The data bases must remain accessible even though targets malfunction. Coping with such failures is not an easy task. Furthermore, efficiency suffers when more and more coping takes place. A graceful degradation is required which departs from the point-to-point orientation between host and target. To *illustrate the problem Figure* 3-4 *presents* a *redundant* DBMS scattered throughout a network composed of four targets. Jobs submitted to the network have been segmented. Auxiliary Directories are provided for each network job for backup purposes. Each directory associates itself with a particular host and documents the source of that host's segments. In case that host goes down, a backup Auxiliary Directory is provided on an alternative host. This duplicate directory can be used to regenerate the original host's Auxiliary Directory when it returns to the network. The threesome composed of the DBMS job, the Auxiliary Directory, and the Auxiliary Directory Backup, address network tasking but ignore the distributed data base problem. Consequently, Figure 3-4 includes DBMS files associated with each host and provides redundant backup. That backup resides on a different host within the network, i.e., the same host providing the Auxiliary Directory backup. Under such an architecture the ingredients for graceful degradation of both the network operating system and distributed data bases are evident. When such degradation is perceived becomes important. Obviously Auxiliary Directories and their backups should be updated coincidentally. Point-to-point communication does not achieve such coincidence. When Host A originates a segment for Host B it updates the Auxiliary Directory for Host B as well. However, updating of the duplicate Auxiliary Directory on Host C should be done coincidentally. In point-to-point schemata Host A updates Host B and then Host C.

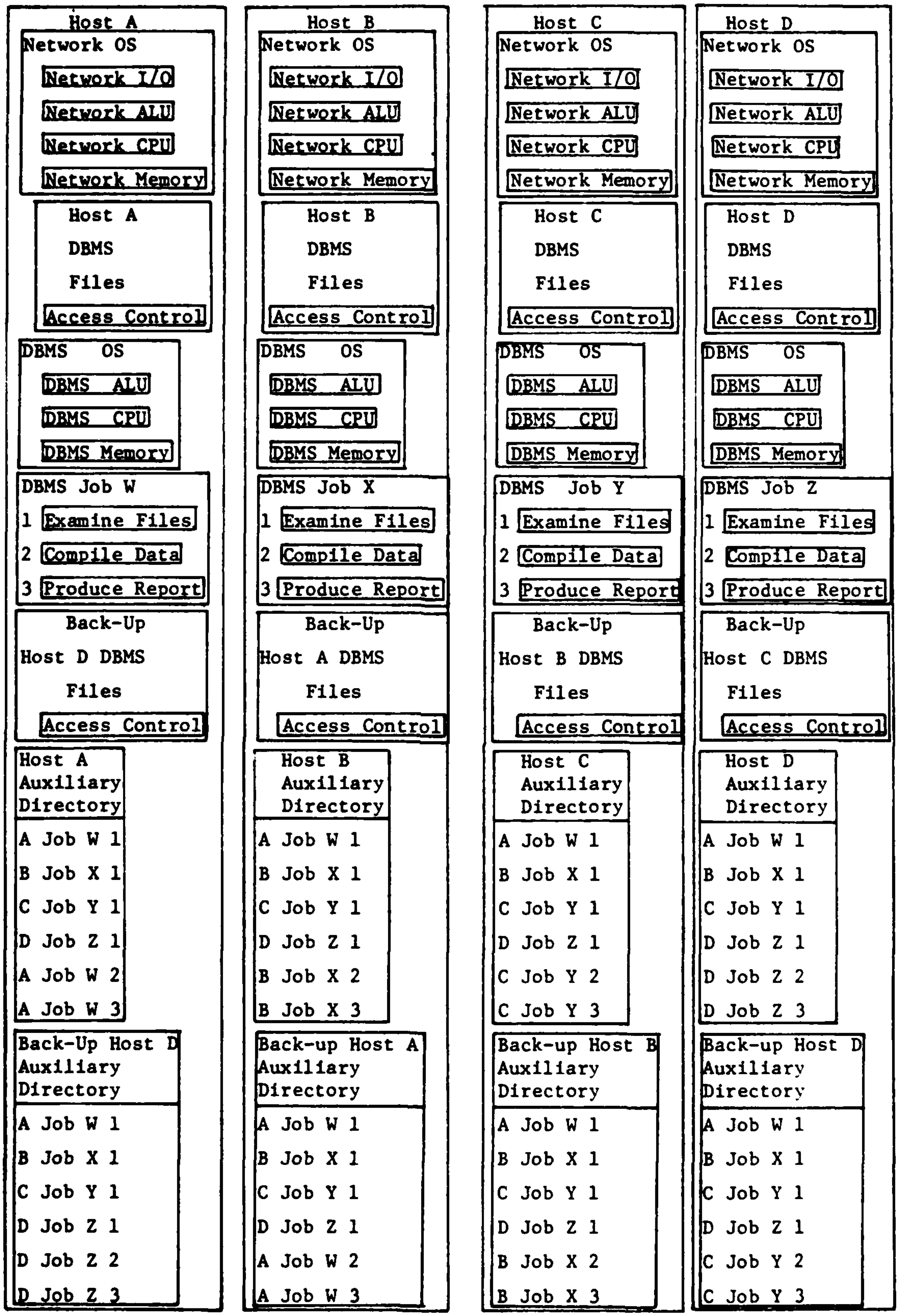


Figure 3-4 DBMS Job Segmentation and Redundant DBMS File Storage

The same observation can be made of the distributed data base. In both instances, a reliable broadcast approach must be implemented.

The structure behind DBMS jobs is presented in Figure 3-5. Note the tight-coupling of examinations to the DBMS files controlled by each host. Whereas individual hosts can compile and produce reports, the very tight coupling across all partitions is required within truly decentralized data bases. Under circumstances of graceful degradation, some of the nodes within Figure 3-5 will disappear. How long their disappearance is tolerated and what happens while they are gone is a decentralized data base problem. It involves failure detection, partitioning, and the operating around missing nodes. Tools to develop such architectures and software systems are not readily available. File allocation schemes within distributed DBMS approaches lack generality. User demand for joining the relations between two targets is not being addressed. Complete synchronization with sufficient redundancy is also not being addressed. Finally, tools to reassign DBMS files to different hosts and locations within network operating systems are not sufficiently general in their scope.

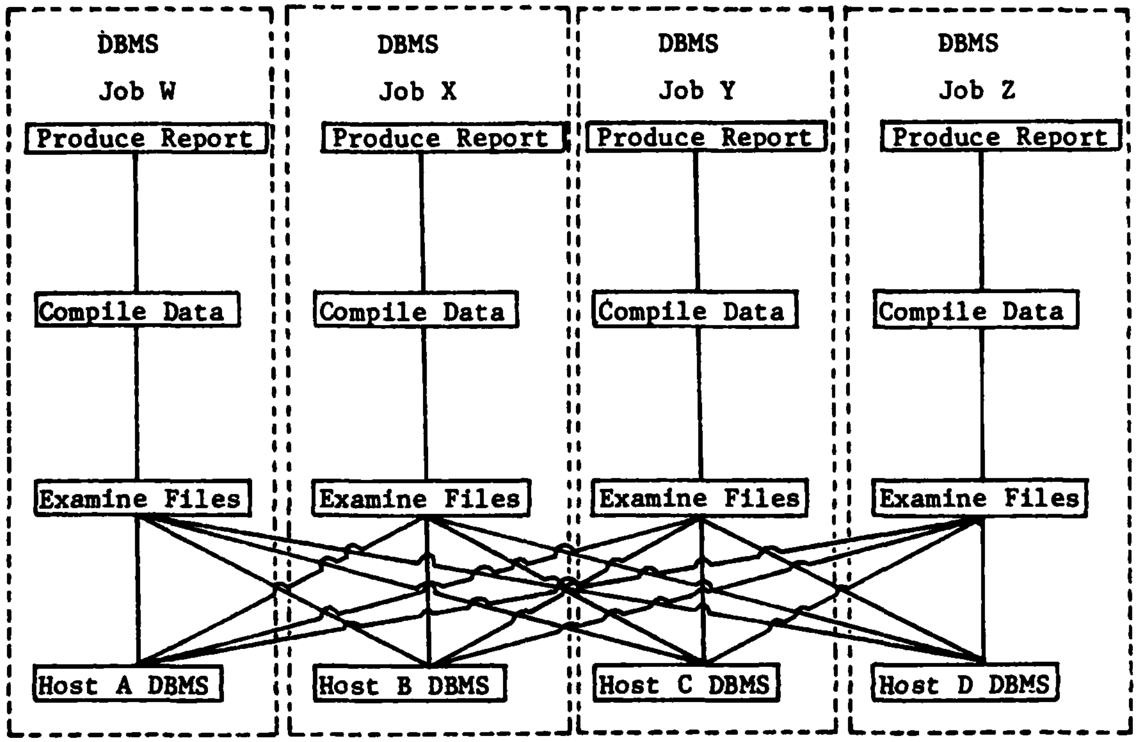


Figure 3-5 Hierarchical Structure of the DBMS Jobs

3.2.4 Intelligence in Environments

The term "intelligent" has begun to appear in discussions of automated environments. While the term often signifies merely a high degree of functional automation, it increasingly refers to a set of characteristics bearing on issues of adaptability, action on incomplete information, heuristics for search and evaluation, and organized knowledge of application domains and of programming language rationale.

Exploration of these issues in depth has implications for far-term generic techniques in distributed processing. Therefore, most of the discussion of sophisticated artificirlly intelligent tools in this report appears in Section 5. Nevertheless, there is a place in the present section for a preliminary discussion of intelligence in support environments which will serve as a bridge between near-term design and development considerations on one hand and far-term tools and techniques on the other. In this section we mainly want to point out that near-term tool definition should consider ease of integration with intelligent systems of the future.

Although intelligent development support systems may be designed essentially independently of evolving concepts of intelligence in the operational systems, both systems will employ the same generic technology. For example, in the design of an application system, there likely will be an integrated intelligent toolkit to synthesize strategies for an intelligent application (e.g., decomposition of a task definition and assignment to distributed components). One implication for near-term tool definition is that the tool's potential support for or enhancement to intelligent systems should be addressed. For example, it will be desirable to address the tool's potential support for the type of knowledge base that will be the foundation of future intelligent systems. Further, it will be profitable to consider the relation of support tools to the predictable characteristics of evolving intelligent run-time support systems for application programs.

4.0 Near-Term Generic Tools

This section of the report serves to correlate information known about existing software tools and the software development life cycle, as well as to propose tools which will be needed in the near future. The tools proposed will provide support to new projects in the area of distributed processing, a complex and evolving aspect of computer science.

Two basic approaches were available to identify requirements for new tools. The first approach would be to survey all tools; to identify the functionality of each tool with respect to the life cycle; and to determine if any aspect of the life cycle had not been addressed. This approach was not chosen primarily because of the unnecessary work that would be performed in evaluating the capabilities of functionally redundant tools.

Additionally, the task's complexity would be high because the search would entail looking for a functional characteristic that is a member of the set of all characteristics of software without knowing what the members of the set were.

The second approach consisted of a chronological reversal of the first approach. The characteristics of software are used as the basis for investigation of tools. Knowing the characteristics of software, a single (not all) tool can be found which enhances or evaluates those characteristics. If no tool can be found, that characteristic becomes a basis for tool development. The redundancy of the first approach would disappear because only a single tool-to-characteristic evaluation would be required; and the complexity of the first approach would be decreased because the starting point would be a known set of characteristics. Attempts will not be made to evaluate criteria that might be decreased when one or more other criteria are enhanced. Obvious relationships will be identified, but extensive evaluation will not occur.

4.1 Definition of Criteria and Life Cycle Phases for Software

This section contains definitions of the criteria for judging software characteristics which we have chosen, and the software life cycles to which they apply. The criteria were chosen after a study of the criteria defined in RADC-TR-80-109, "Software Quality Metrics Enhancements", by General Electric, and a slide presentation made by Boeing at the Distributed Processing

Technology Exchange Meeting at RADC on 18-20 May 1982. The list of criteria developed by these companies was much longer and more detailed than required for the purposes of this report, so categories of criteria were combined and deleted, and new definitions were written for those which were left. The life cycle phases used in this report are also a somewhat smaller set than sometimes used, since it was felt that this less detailed breakdown was more in keeping with the needs of the report.

4.1.1 Software Life Cycle Phases

The development of software requires that it progress through a life cycle consisting of requirements, design, coding, testing, maintenance and operation. The first four phases are concerned with the creation of the software. The latter two phases are concerned with the quality and reliability of the existing software. The following subsections are concerned with the definition of each of the life cycle phases.

4.1.1.1 Requirements Phase

The specification of system requirements is the first step in the software development life cycle. This phase begins with the statement of a problem to be solved and ends with a specification of what the system to solve the problem must look like. The goal of this phase is to clearly define and document the set of criteria by which a program will be ultimately examined for adherence to the specifications. The specification and documentation of the requirements of a system can be partially automated through the use of software tools. These tools allow the development of requirements specification documents using defined methodologies and analysis of the specifications for data flow and control sequences.

4.1.1.2 Design Phase

The second step in the software development life cycle is to develop an implementation for the previously established requirements. This ideally takes the form of a complete design that provides both an outline of the functional components of the system to be implemented and an explanation of how the requirements specifications will be met using the outlined system. This would additionally provide for precise, accurate and orderly transitions between the requirements design and coding activities. To this end the detail of the resulting design must be sufficient so that an implementors decisions cannot interfere with the ultimate satisfaction of specified requirements.

4.1.1.3 Coding Phase

Following or possibly overlapping the design phase is the implementation (coding & debug) phase. This phase is generally a manual process though defined methodologies do exist to help organize and improve the activity such as structured programming, and bottom-up and top-down implementations.

4.1.1.4 Testing Phase

The testing phase is a validation process that examines the implemented system to insure that the initial requirements are met. This process should additionally include tests to insure the quality and reliability of the system. This has become especially important as systems continue to grow in size and complexity. As with the other life cycle phases this process may overlap the previous (code and debug) phase. The two main elements of testing and quality assurance are static and dynamic testing both of which are characterized by the virtual necessity for the use of automated tools.

4.1.1.5 Maintenance Phase

The maintenance phase is a process continuing throughout the life of the software to ensure quality and reliability. Maintenance begins when changes in the software are required by management or when errors are found by the user during the operation of the software. When maintenance begins, it may require additional requirements, design, coding and testing. Since 75% or more of the time is spent in the maintenance phase, this phase is critical in the life cycle of software. It necessitates that effective tools exist to aid the software support personnel to provide timely and effective maintenance.

4.1.1.6 Operations Phase

The operations phase is that user-oriented phase in which software performs its planned and required function. With the aid of documentation and error-reporting tools, the user is provided the capability for monitoring and interacting with operatonal software to assure intended functionality is reached. When errors occur, they are either identified as user-originated or reported to software support personnel for correction. The operations phase requires correct documentation and effective error-reporting tools. User capability for monitoring and interacting with operational software is not required for embedded weapon systems. Such capability is more characteristic of communication systems.

4.1.2 Tool Criteria

The following definitions concern criteria for judging software characteristics. The criteria will be used to choose a minimum set of generic tools for the software life cycle phases.

Traceability - A program is traceable (exhibits traceability) if a thread exists to tie the modules of the program back through design to requirements. Traceability can exist independently in two directions: from requirements to the program, and from the program's modules to the requirements. In order to be fully traceable a program must exhibit traceability in both directions. Traceability must include design. That is, whatever design documents were retained as program documentation must be included in the thread.

Consistency - A program exhibits consistency if the requirements, design, and implementation techniques and notation are uniform throughout. Use of standardized languages and techniques are necessary to insure consistency.

Fault Tolerance - A program is fault tolerant if it is capable of operating in a consistent manner in spite of program errors, errors in input data, and hardware malfunctions. Total fault tolerance is an impossible goal, since there are hardware malfunctions and program errors from which no recovery is possible. The degree of fault tolerance which is desired and the actions to be taken under various conditions should be specified in the requirements document.

Simplicity - A program exhibits simplicity if each individual module is coded in an understandable manner, and the modularity has been established with consideration to a specific method, i.e., data structure, control flow, functionality, etc.

Modularity - A program is modular if its structure consists of highly independent modules. A module is independent if it could be implemented in a different manner without affecting the other modules of the program.

Functional Generality - A program or a module exhibits functional generality if its functions are not unnecessarily restricted. One example would be a routine to produce a line of print. If the line length is passed as a parameter rather than being "hard coded", the routine will be more general at a cost of very little additional complexity.

Expandability - A program is expandable if it is easy to add new functions, to enhance its current functions, or to increase the amount or types of data handled.

Instrumentation - Instrumentation provides the user and/or the maintainer with information on the operation of the program. For the user, it generally means status information. For the maintainer, it means such information as how many times a particular function is called, how data is distributed among differing types, and records of type and frequency of errors.

Resource Utilization - Resources utilization is the measure of how well a program conserves system resources. These resources include time, memory and external storage. How much priority is given to the conservation of each of these resources is a function of the requirements of the system.

Control of Data Access - Control of data access reflects two conflicting requirements: ease of access and restriction of access to sensitive data. Program data should be easily accessible to all modules and users who need it.

Ease of access is especially important in distributed systems where the user may be physically distant from the data being accessed. However, modules which do not need a particular datum and users who are not authorized to have specific information should be prohibited from access. In addition, a requirement may be that the system keep records of attempted and successful accesses to sensitive data.

Ease of Use - Ease of use measures the amount of effort which must be put forth to operate the system. It includes simplicity of input preparation, simplicity and understandability of operator commands, understandability of output data, and the amount of training required for new users of the system.

Independence - A program's independence is determined by the extent to which it relies on a specific hardware system or a specific underlying software system (operating system or run time system). A program is more independent when those functions which must be made specific to most hardware or software are isolated in lower level modules or are parameterized to allow easy change during system builds.

Commonality - A program exhibits commonality to the extent that standard interfaces are used between modules and that standard data formats are used. An effect of commonality should be the development of reusable software.

Compliance - A compliant program meets all the requirements laid down for it. This includes, but may not be limited to, normal processing, error handling, response time, memory/resource usage, and the accuracy of results.

Clarity - A program exhibits clarity through its documentation, including its internal documentation, to the extent that that documentation is readable and understandable.

Virtuality - A program exhibits virtuality if the user is not required to have a knowledge of the hardware implementation in order to run the system. Such things as the number and type of auxiliary storage devices, the amount of main storage, and even the type of CPU should be transparent to the user.

Distributedness - Distributedness is the extent to which elements of the system are logically and/or geographically separated. The word elements as used above includes both software and hardware. Software specific considerations are distribution of control, interconnect architectures, and data bases.

4.1.3 Correlation of Life Cycle Phases and Criteria

The correlation between life cycle phases and the criteria is given by Table 4-1. Almost all criteria apply to the initial phases from requirements through coding. The criteria that do not apply to requirements (simplicity, modularity, and functional generality) all deal specifically

		SOFTWARE QUALITY CRITERIA																
		TRACEABILITY	CONSISTENCY	FAULT TOLERANCE	SIMPLICITY	MODULARITY	FUNCTIONAL GENERALITY	EXPANDABILITY	INSTRUMENTATION	RESOURCE UTILIZATION	CONTROL OF DATA ACCESS	EASE OF USE	INDEPENDENCE	COMMONALITY	COMPLIANCE	CLARITY	VIRTUALITY	DISTRIBUTEDNESS
SOFTWARE LIFE CYCLE PHASES	REQUIREMENTS	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	DESIGN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	CODING	✓	✓		✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓
	TESTING	✓							✓	✓					✓			✓
	MAINTENANCE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	OPERATIONS			✓					✓	✓	✓	✓			✓		✓	

Table 4-1 Life Cycle Phases vs. Software Quality Criteria

with design and coding. Criteria that do not apply to coding (fault tolerance, expandability, instrumentation and control of data access) are areas which have already been settled by design and requirements. The testing phase is concerned with measuring compliance, which includes resource utilization. This is accomplished with the use of instrumentation. Traceability and distributedness must be maintained through this phase. The operations and maintenance phases affect all criteria.

4.2 Correlation of Tools to Criteria

This section contains a list of generic tools with their definitions. Each tool is related to the life cycle phase of its primary use. Criteria enhanced by each tool during its primary life cycle phase are discussed (refer to Table 4-2).

Table 4-2.

Software Quality Criteria

vs.

Software Tools

		SOFTWARE TOOLS (by Life Cycle Phase)																																			
		Requirement					Design			Coding							Testing																Maintenance			Operations	
		AUTOMATED REQUIREMENTS DOCUMENT GENERATOR	AUTOMATED REQUIREMENTS ANALYZER	AUTOMATED LANGUAGE TRANSLATOR	REQUIREMENTS INTERFACE PROCESSOR	RAPID PROTOTYPER	CALLING TREE GENERATOR	CROSS REFERENCE GENERATOR	STRUCTURE CHECKER	OPTIMIZING COMPILER	CROSS COMPILER	LINKER/LOADER	CHECK-OUT COMPILER	STANDARDS AUDITOR	FORMATTER	MENU GENERATOR	COMPLETION ANALYZER	STUB GENERATOR	MUTATION TESTER	PATH FLOW ANALYZER	STORAGE DUMP	ASSERTION CHECKER	RESOURCE MANAGEMENT ANALYZER	CORRECTNESS ANALYZER	SYMBOLIC DEBUGGER	HISTORICAL FILE GENERATOR	INTERFACE MAPPER	VARIABLE MAPPER	CONNECTIVITY ANALYZER	REACHABILITY ANALYZER	TIMING ANALYZER	USAGE COUNTER	VERSION GENERATOR	RAPID RECONFIGURATOR	REPORT GENERATOR	DIAGNOSTIC ANALYZER	SYSTEM BUILDER
Software Quality Criteria	TRACEABILITY			+																		+		+									+				
	CONSISTENTY		+	+										+	+							+		+													
	FAULT TOLERANCE				+																													+		+	
	SIMPLICITY										+			+	+							+												+			
	MODULARITY											+															+		+	+							+
	FUNCTIONAL GENERALITY				+													+																			
	EXPANDABILITY	+			+																												+				
	INSTRUMENTATION				+						+		+				+		+	+	+			+	+	+		+			+	+					
	RESOURCE UTILIZATION				+					+	+	+											+								+			+		+	
	CONTROL OF DATA ACCESS																																+				
	EASE OF USE				+											+																			+		
	INDEPENDENCE																																				
	COMMONALITY			+								+															+							+			
	COMPLIANCE		+		+	+				+	+	+		+			+		+	+		+	+	+	+						+						
	CLARITY		+				+	+	+													+											+				
	VIRTUALITY				+																																
	DISTRIBUTEDNESS				+																																

4.2.1 Requirements Tools

4.2.1.1 Automated Requirements Document Generator

This tool generates a requirement by accepting an implementation independent specification couched in a formalized language, performs processing to format requested output, and then generates the output in graphic or textual representations as required. Expandability is enhanced when new requirements are generated during the maintenance phase.

4.2.1.2 Automated Requirements Analyzer

Assuming the requirements are specified in a formalized language, this tool will provide checks for completeness, consistency, and redundancy of information given. Compliance is enhanced since the completeness of the specification is checked, and clarity is enhanced by not allowing redundant names or functions.

4.2.1.3 Automatic Language Translator

One of multiple languages is selected to be the target language for the implementation independent specification. This tool then automatically translates the specification into a compilable program in that language. Traceability, consistency, and commonality are all enhanced by this tool. Simplicity and clarity would probably be decreased by this tool.

4.2.1.4 Requirements Interface Processor

This tool is a front end to the automated requirements document generator. Its function is to provide user-friendly access and use of the tool as well as provide a powerful modification capability to support rapid prototyping and simulation. Multiple data entry techniques, such as graphic representation, menu representation, default selection, etc. should be support d. Criteria enhanced indirectly include ease of use, resource utilization, compliance, fault tolerance, functional generality, expandability, instrumentation, virtuality, and distributedness since these may be modeled during development of the requirements specification (assuming an analyzer and translator are present). No criteria are directly enhanced by this tool since the final product could achieve the same status through manual methods of analysis and documentation over an extended period of time.

4.2.1.5 Rapid Prototyping

This process involves automating the labor intensive portions of feasibility studies through computer simulation and modeling techniques.

4.2.2 Design Tools

4.2.2.1 Program Design Language (PDL) Calling Tree Generator

A calling tree generator is similar in operation to a cross reference generator, but is restricted to subprogram names rather than all identifiers. The output is arranged in the opposite manner to that in which cross reference shows where a particular name is used, that is, that C is called by A and B, while the calling tree shows that A calls B and C. This different orientation makes a valuable addition to documentation, increasing clarity.

4.2.2.2 PDL Cross Reference Generator

A cross reference generator accepts as input a design expressed in PDL and produces as output a listing of all points of definition and all references which can be retained as documentation, thus improving clarity.

4.2.2.3 Structure Checker

A structure checker for PDL is a program that accepts as input a design expressed in PDL and produces as output the same PDL with its structures checked for correctness and completeness. Structure here refers to programming language structures such as if-then-else, begin-end and the case statement. Since PDL may be kept as documentation for the finished product, clarity is improved by this process.

4.2.3 Coding Tools

4.2.3.1 Optimizing Compilers

Optimization is a compiler function which improves the quality of the machine language code produced by the compiler. Most optimizations improve both speed of execution and the size of the executable program. They therefore improve resource utilization. In extreme cases of memory size limits or response time constraints, optimization may be required for compliance. Six common optimizations are: constant propagation, common subexpression elimination, strength reduction, code motion, dead code elimination, and the elimination of induction variables. Constant propagation occurs when the optimizer recognizes computations for which all data is available. Since the data is available, the compiler can do the computation, providing its result as a constant. Therefore, the time and code required to compute the result during execution is saved. Common subexpression elimination takes place when the compiler recognizes that the same result will be computed in two or more places. (To derive the same result, not only must the expressions be the same except possibly for commutivity, but the data must be the same.) The compiler saves the result from the first computation, and uses it to replace the code required for the subsequent computations. Strength reduction occurs when the compiler is able to replace an arithmetic operation with another operation which requires less time. A good example would be the replacement of X squared by X times X. Dead code is any part of a program which will never be reached during execution. This sort of code is sometimes created by an if statement whose test always has the same result. If the compiler can detect this situation, it can eliminate the if test and the unreachable code, saving both time and memory. Code motion and the elimination of induction variables are both loop optimization techniques. The compiler can often find expressions whose result is the same for each iteration of the loop. These expressions are called loop invariants. Code motion is the removal of these expression from within the loop, and their placement just before the loop, where they will only be executed once. Induction variables are variables whose values vary in a linear fashion during execution of the loop. Not all induction variables are part of the source code. A FORTRAN example might be:

```
         REAL X(10), Y(10)
         DO 10,I=1,10
         X(I)=0
    10   Y(I)=0
```

There are three induction variables in this code segment: I and the offsets used to address elements of the arrays X and Y. If a real occupies 4 address units of storage, the expression for the offset into X is (I-1)*4. The expression for Y is the same. A good compiler might first recognize these two as the same and eliminate the second computation. (common subexpression elimination). It might then realize that it could get the same result by setting the offset to zero outside the loop and adding four to it on each loop pass. (strength reduction). Finally, it might realize that it does not need both the offset and I, since it can determine loop termination by testing the value of the offset. Therefore, it could eliminate I from the loop (elimination of an induction variable). Two types of loop optimization which are rarely used are loop unrolling and loop jamming. Loop unrolling can only be done when the number of times a loop will be executed is known. It consists of duplicating the body of the loop a number of times in order to eliminate some of the tests for loop termination. This technique saves time, but invariably wastes space. Loop jamming can take place when two loops have the same indices and no result of the first loop is used in the second. The two loops are merged into one. A (trivial) example might be:

```
        REAL X(10),Y(10)
        DO 10,I=1,10
    10  X(I)=0
        DO 20,I=1,10
    20  Y(I)=1
```

Which might be merged into:

```
        REAL X(10),Y(10)
    DO 10,I=1,10
        X(I)=0
    10  Y(I)=1
```

All of the above optimizations save time. Most of them also save space, with the exception of loop unrolling and possibly code motion.

4.2.3.2 Cross Compiler

A cross compiler is a compiler which is hosted (runs) on a type of machine different from that for which it generates code. Usually this means that the host machine is a mainframe or a mini computer and the target machine is a microprocessor. Cross compilers allow the use of larger, more complex compilers (which may be required for complex languages such as Ada or PL/I) than could be hosted on the target. The features of the cross compiler can then include optimization and/or the options of a checkout compiler. Therefore cross compilers may indirectly improve instrumentation, resource utilization, and compliance, while also improving ease of development. A cross compiler makes it possible to use a more complex compiler (or a compiler for a more complex language) than might be available on the target system. It may improve resource utilization, simplicity, and compliance.

4.2.3.3 Linker/Loader

The normal functions of a linker/loader are to allow the usage of external routines and to allow programs to be loaded in different locations (relocation). These functions increase commonality, and modularity. In addition, some linker/loaders may provide facilities for overlaying program segments. Overlaying allows different program segments to occupy the same memory locations, with each segment being read in as it is needed. This improves one facet of resource utilization (space) at the expense of another (time). It may be required for compliance.

4.2.3.4 Checkout Compiler

Checkout compilers provide special services during the compilation of programs which assist during program checkout. Options in the compiler provide for automatic printing of variables each time their values change, for automatic trace of subprogram calls, or for collection of other statistics such as the amont of time spent in each routine. In other words, a checkout compiler adds instrumentation.

4.2.3.5 Standards Auditor

A standards auditor (code auditor) takes as input a source program in some specific language and produces a report detailing violations of some set of programming standards. By enforcing standards, it improves consistency, simplicity, and clarity. (Clarity is improved because the use of procedure headers may be part of the standard being enforced.)

4.2.3.6 Formatter

A formatter is a program which takes as input a source program in some specific high order language and rearranges the input source into some specified format, enforcing standard indentation conventions and other standards for layout of the program on the printed page. It therefore improves the consistency of the program. Since understandability is improved by proper indentation, the simplicity of the program is also improved.

4.2.3.7 Menu Generator

A menu generator is designed to provide optimum usefulness and versatility in data entry by utilizing video display terminals. A user-defined form, or mask, for data manipulation on the display area improves ease of use. Since the mask resembles a printed form, data is placed into the form by filling in the appropriate blanks on the screen.

4.2.4 Testing Tools

4.2.4.1 Completion Analyzer

A completion analyzer (or coverage analyzer) provides data that shows how thoroughly the source code has been exercised during the testing with respect to the testing goals which provides compliance and adds instrumentation.

4.2.4.2 Stub Generator

A stub generator provides substitutes during testing for modules which have not been coded. Testing of individual modules is thus made much easier. This tool enhances functional generality.

4.2.4.3 Mutation Tester

A mutation tester constructs a set of mutants of the target program which will test a program's compliance. A mutant is a program statement which has been transformed in such a way as to effect typical program errors. A programmer could test a program with the assumption that the current state of the program is a mutant of the correct one.

4.2.4.4 Path Flow Analyzer

A path flow analyzer is a software technique which provides instrumentation and compliance by scanning the source code in order to design an optimal set of test cases which exercise the primary paths in a software module.

4.2.4.5 Storage Dumps

Storage dumps provide program/system status and selected data values which contribute to instrumentation.

4.2.4.6 Connectivity Analysis

Connectivity analysis is used to identify the direct program paths between any two sections of code within a program, segment tracing, which provides a measure of modularity of the program.

4.2.4.7 Reachability Analysis

Reachability analysis is used to identify the specific program paths, direct or indirect, exercised in order to reach a specific module, subroutine or section of code within a program which provides a measure of modularity of the program and distributedness of the system of programs. It can also be used to identify unreachable modules and "dead" code.

4.2.4.8 Timing Analyzer

A timing analyzer reads the executable code and produces a report showing program segment invocation hierarchy and the actual execution times per complete program segment cycles. Instrumentation, resource utilization, and compliance are measured with respect to the timing in the operation of the program.

4.2.4.9 Symbolic Debugger

A symbolic debugger is used to enhance instrumentation. Since testing is the process of determining whether or not errors or faults exist in a program, debugging is an attempt to isolate the source of a problem and to find a solution with snaps of variables and absolute identifiers which enhances compliance.

4.2.4.10 Historical File Generation

Historical file generation provides instrumentation by the generation of accumulated execution statistics for all test cases including blocks executed, paths taken, modules invoked, etc.

4.2.4.11 Interface Mapping

Interface mapping is used to measure commonality with respect to the identification of program interfaces such as called and calling modules or modules involved in interprocess communications and the verification of the range and limits of the module parameters. It is useful for the analysis of modularity, a module's impact on other modules, and the identification of data abstractions from subroutine calls, function calls and macros.

4.2.4.12 Variable Mapping

Variable mapping provides information with respect to the definition and use of the individual variables in the program and may provide actual values and initialization, during execution of the program. Instrumentation is measured with respect to the information provided.

4.2.4.13 Assertion Checker

An assertion checker is used to check a program's critical requirements for compliance with the results derived by dynamic analysis. It enhances the simplicity and maintainability of the program and improves the traceability and consistency of the software. By inserting assertions concerning the value or condition of program variables in the program code, assertion checking may be applied to error detection activities, understanding program behavior and clarity through documention of the program's critical requirements.

4.2.4.14 Resource Management Analysis

Resource management analysis dicatates resource utilization for the purpose of compliance in processing requirements such that programs and data should be allocated the minimum amount of time and storage that is necessary. When additional amounts are needed, they are acquired and released dynamically.

4.2.4.15 Correctness Analyzer

A correctness analyzer determines the traceability between a program's total response and the stated response in the functional requirements and between the program as coded and the programming requirements. Measuring the program's response helps to determine instrumentation and compliance.

4.2.4.16 Usage Counter

A usage counter reads the executable code and collects usage data during program execution such as the number of times each executable statement, branch and subroutine calls were executed. Instrumentation is measured with respect to how many times a particular statement, branch or subroutine is executed.

4.2.5 Maintenance Tools

4.2.5.1 Version Generator

A version generator is a system to track and control changes to files (source, object or text) associated with software development which enhances the control of data access, provides expandability, and ensures traceability in the maintenance life cycle. The system should be able to store, update and retrieve files including audit trails as well as maintain historical records on all versions controlled for the purpose of complete program documentation and clarity.

4.2.5.2 Rapid Reconfiguration

Rapid reconfiguration is an automated process by which a system is rebuilt after changes which provides fault tolerance. All file dependencies and processes (resource utilization, common memory, compilations, preprocessing, etc) are specified in a hierarchical manner such that a change in one module can easily and quickly be related to changes required in other modules which provides commonality and simplicity.

4.2.5.3 Report Generator

A report generator consists of methods for customized formatting of generated output to provide ease of use and improve resource utilization.

4.2.6 Operations Tools

4.2.6.1 Diagnostic Analyzer

A diagnostic analyzer measures the capability of the system to perform its functions in accordance with design requirements, even in the present of hardware failures. If the system functions can be performed in the event of faults, the system is partially fault tolerant when design specifications are not met with respect to the time required or the storage capacity required to complete the job. Fault tolerance is provided by the use of redundant resources, resource utilization, for upgraded system reliability and protection.

4.2.6.2 System Builder

A system builder identifies all required programs, and it compiles, links, relocates and produces an object file for execution which enhances modularity.

4.3 Recommended Near-Term Tools and Their Flowcharts

There are five tools which affect the requirements phase. They are the Automated Requirements Document Generator, the Automated Requirements Analyzer, the Automatic Language Translator, the Requirements Interface Processor, and the Rapid Prototyper. The Automatic Language Translator directly enhances traceability, consistency, and commonality. The Automatic Requirements Analyzer enhances consistency and compliance. All other criteria influenced are influenced indirectly (see Table 4-2).

The design phase, in which decisions are made which affect the most criteria, is almost devoid of automated tools. There exist several useful methodologies which were developed mostly for use in the business data processing sphere, but none of them have been satisfactorily automated. Some of the tools discussed for the requirements phase extend into the design phase, but the act of design still remains a manual (or even cerebral) art. The tools which apply to this phase which will be discussed here all fall into the class of program design language processors, and are generally found as a single tool rather than individually. All three tools (a calling tree generator, a cross reference generator, and a structure checker) are concerned with the production or checking of documentation, therefore the only criterion affected is clarity.

Tools for the coding or implementation phase can be divided into two areas: tools affecting source code, and tools affecting object code. The tools affecting source code are the menu generator, the standards auditor, and the formatter. The menu generator helps coders to lay out and design screens and menus for interactive input. By doing this it affects ease of use of the finished project. Both of the other tools affect consistency and simplicity of the source code, and the standards auditor also affects compliance if coding standards are specified in the requirements. The tools affecting object code are the optimizing compiler, the cross compiler, the linker/loader, and the checkout compiler. The first three of these all affect compliance and resource utilization by making programs more time and/or space efficient. The cross compiler also affects simplicity and possibly instrumentation if it has some of the features of the checkout compiler. The linker/loader affects commonality and modularity as well. The checkout compiler adds instrumentation only.

During the maintenance phase, all decisions made prior to this phase are either maintained or changed with respect to errors detected in either operations or requirements and design changes requested by management. Maintenance is accomplished with a generic maintenance tool comprised of a set of specific maintenance tools which provide correction of errors and requested changes in documentation and programs by modification, addition or removal of functions. Additionally, tools must be provided after error correction or requested changes. Replication of programs and documentation is ensured by the specific maintenance tools providing redundancy in distributed processing systems.

The criteria relating to redundancy in requirements and documentation are traceability, commonality, and clarity. The criteria relating to maintenance activities such as error correction are resource utilization, control of data access and fault tolerance. Since approximately 75% of the software life cycle is devoted to maintenance, it is necessary for a generic maintenance tool to enhance the aforementioned criteria as well as enhance ease of use, expandability and simplicity.

With reference to Table 4-2, the aforementioned criteria are enhanced by the version generator and rapid reconfiguration. Version generation provides an audit trail for maintenance of programs and their data dependencies. Using this audit trail, rapid reconfiguration automates rebuilding of a system. Additionally, the report generator enhances ease of use during the maintenance of programs. The combination of these specific maintenance tools forms a minimum generic maintenance tool.

During the testing phase, compliance and instrumentation are the important criteria measured by the specific testing tools as shown in Table 4-2. In order to determine the specific testing tools which are part of a generic testing tool, tools which enhance either or both of the criteria, compliance or instrumentation, should be considered.

The main group of specific testing tools which may be a part of a minimum generic testing tool should enhance both compliance and instrumentation. This group consists of a completion analyzer, mutation tester, path flow analyzer, correctness analyzer, timing analyzer and symbolic debugger. As shown in Table 4-2, these specific testing tools also enhance other criteria such as traceability and consistency in the correctness analyzer and resource utilization in the timing analyzer. The additional criteria improve the minimum generic testing tool which enhances compliance and instrumentation.

A secondary group of specific testing tools which may be a part of a generic testing tool are those which enhance either compliance or instrumentation, but both of the criteria are not enhanced. The tools which enhance compliance, not instrumentation, are an assertion checker and resource management analyzer. Similarly, the tools that enhance instrumentation, not compliance, consist of a usage counter, historical file generator, variable mapper and storage dump (see Table 4-2). Additionally, the assertion checker enhances traceability, consistency, simplicity, and clarity, and the Resource Management Analyzer enhances resource utilization. The additional enhancements further improve the generic testing tool.

A generic testing tool is composed of its minimum requirements if it consists of the main group of specific testing tools which enhance both instrumentation and compliance. The tool is improved if the secondary group of specific testing tools, enhancing either instrumentation or compliance, is added. A combination of the main and secondary groups of specific testing tools form a generic testing tool during the testing phase.

During the operations phase, programs must be capable of operating in a consistent manner with recovery from hardware malfunctions and program errors unless there is no recovery; therefore, a generic operations tool should enhance fault tolerances. Additionally, resource utilization and modularity should be enhanced since programs should conserve system resources. The specific operations tools comprising the generic operations tool should enhance the previously mentioned criteria.

Referring to the specific operations tools in Table 4-2, a diagnostic analyzer and a system builder form a minimum generic operations tool. A system builder, linking all of the programs in a system for execution, enhances modularity. Since a diagnostic analyzer measures the capability of the system of programs to perform its functions in accordance with design requirements, it enhances resource utilization and fault tolerance. Since these specific operations tools enhance the prescribed criteria, they form a minimum generic operations tool.

Compiler Generation Tools

The following too., are somewhat restricted in their application, although the first two could be used to read and parse other forms of command input. Their primary application, though, is the construction of compilers.

Lexical analyzer generators accept a description of the base elements of a language, which are called tokens, and generate tables to be used by a standard program in the recognition of tokens. A token is a sequence of characters which can be treated as a single logical entity. Tokens include keywords such as IF and GOTO, numbers, identifiers and special symbols such as = or <=. The lexical analyzer reads characters until it has recognized a token and then returns the type and value of the token.

Syntax analyzer generators accept a description (called a grammer) of a language and produce tables for use by a syntax analyzer. Syntax analyzers accept tokens from the lexical analyzer and parse the language into larger constructs. This action is somewhat analogous to the actions of an English student in diagramming a sentence. For example, if the lexical analyzer returned the tokens "article", "adjective" and "noun", then the syntax analyzer would recognize that these tokens comprise a "noun phrase". At a later phase the "noun phrase" might be combined with a "verb phrase" and another "noun phrase" into a "transitive sentence". In terms of a programming language, this means that a syntax analyzer would combine "identifier", "assign", "identifier", "plus" and "identifier" into "assignment statement". The compiler then takes this information to generate an intermediate representation of the program.

Code generator generators take as input some form of a *description of a target machine and a description* of the intermediate representation mentioned above. They produce as output tables and/or code for use by a standardized code generator. The code generator takes the intermediate representation of a program and converts it into target machine language. These tools are currently not fully developed, although some industrial use has taken place.

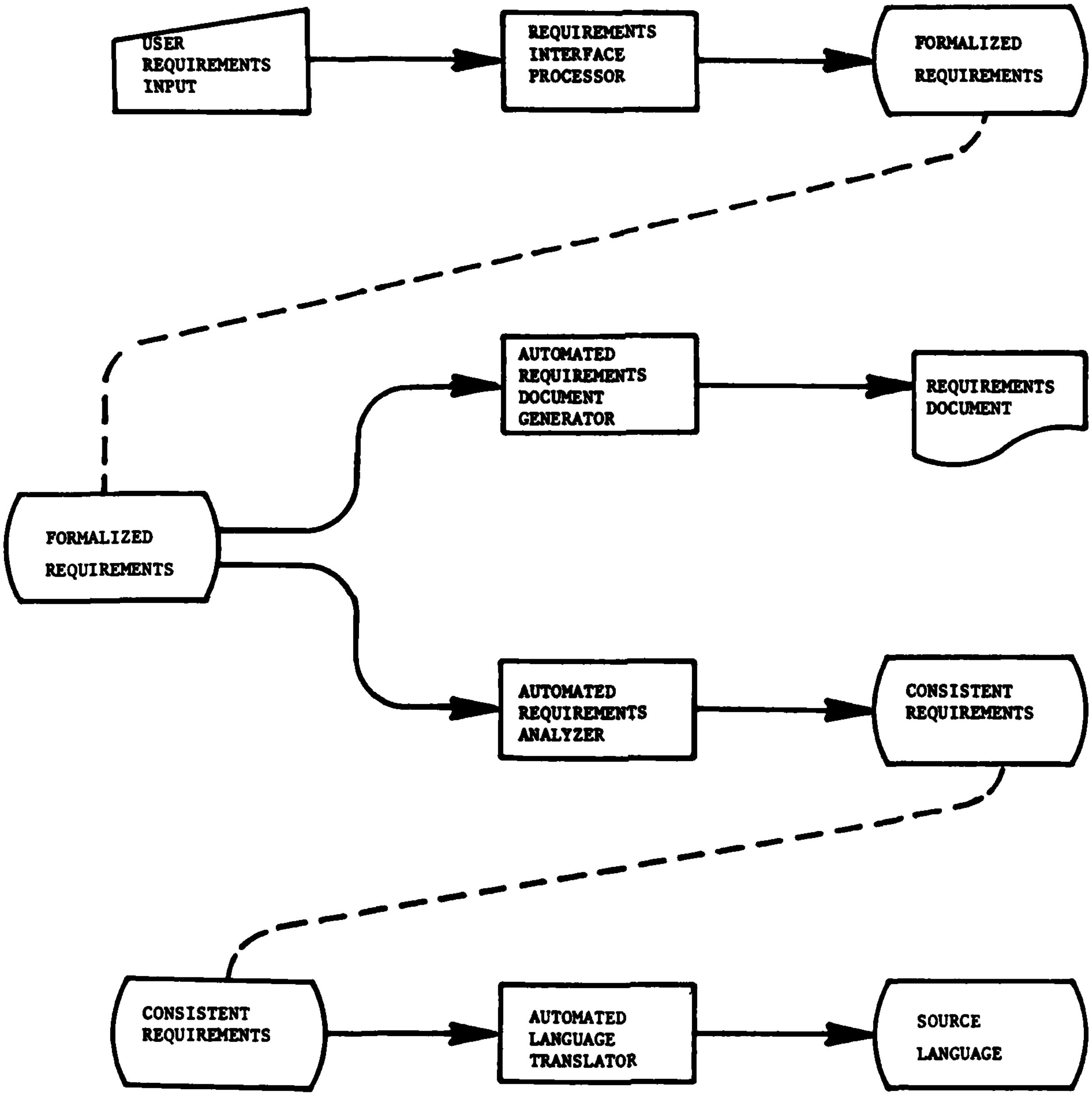


Figure 4-1 Generic Requirements Tool

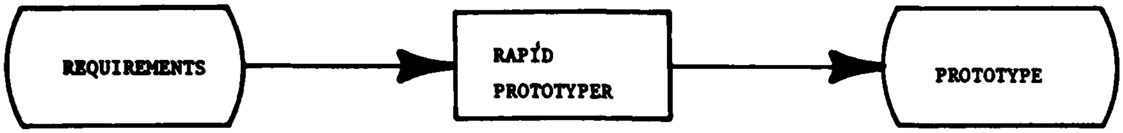


Figure 4-2

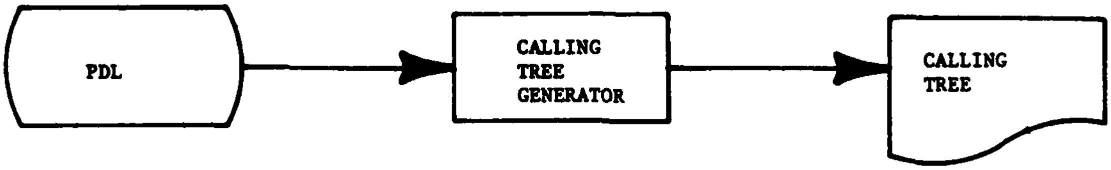


Figure 4-3

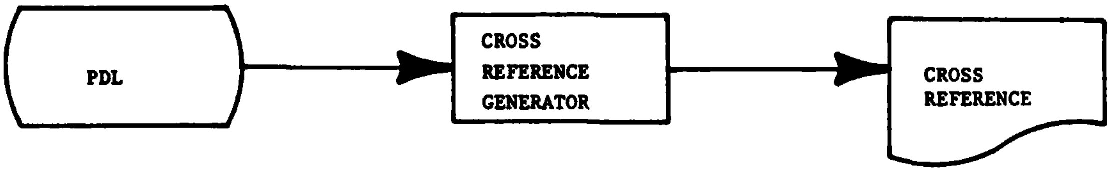


Figure 4-4

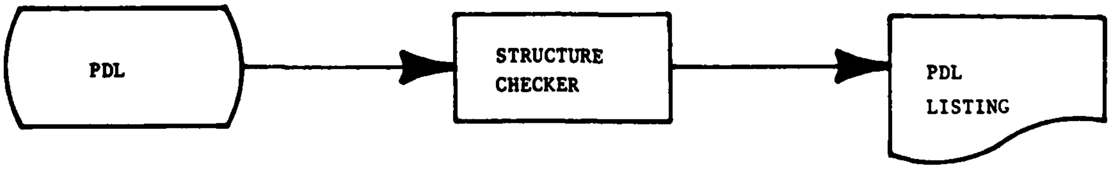


Figure 4-5

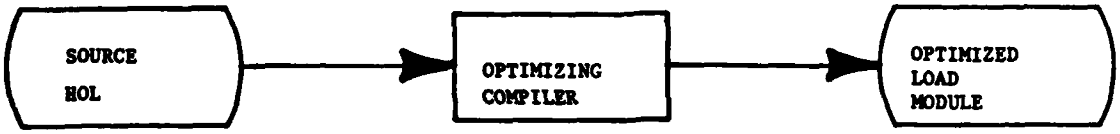


Figure 4-6

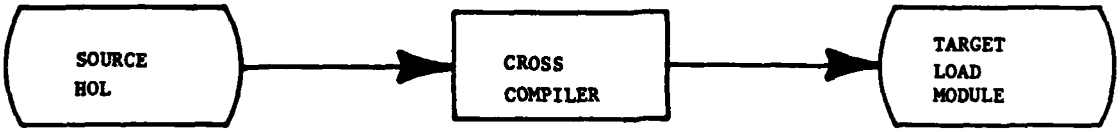


Figure 4-7

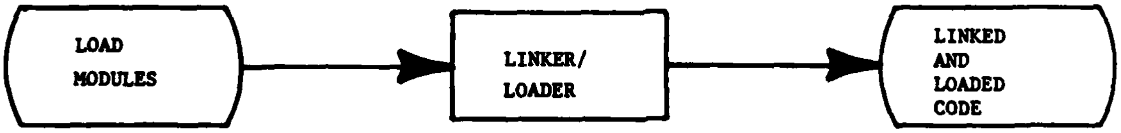


Figure 4-8

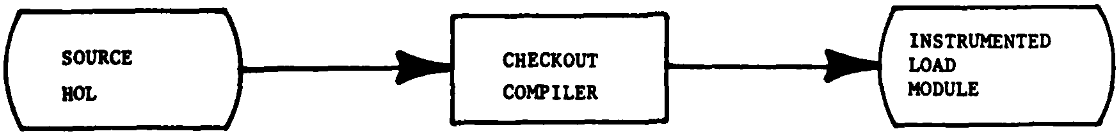


Figure 4-9

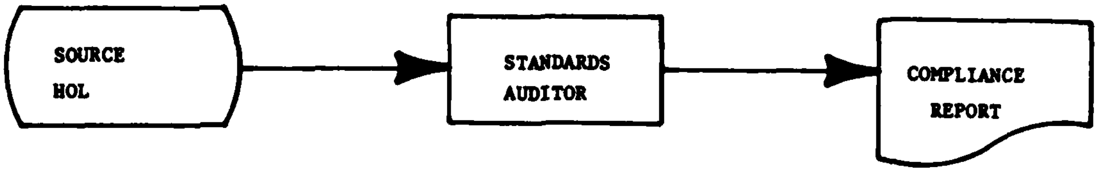


Figure 4-10

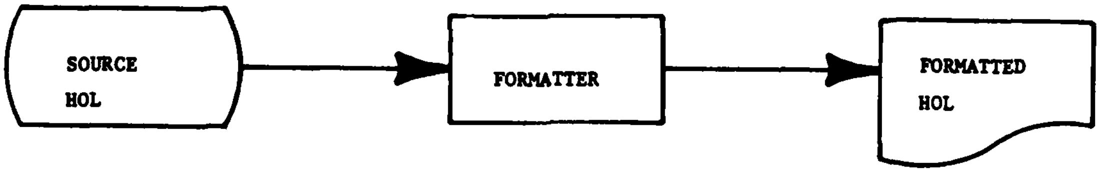
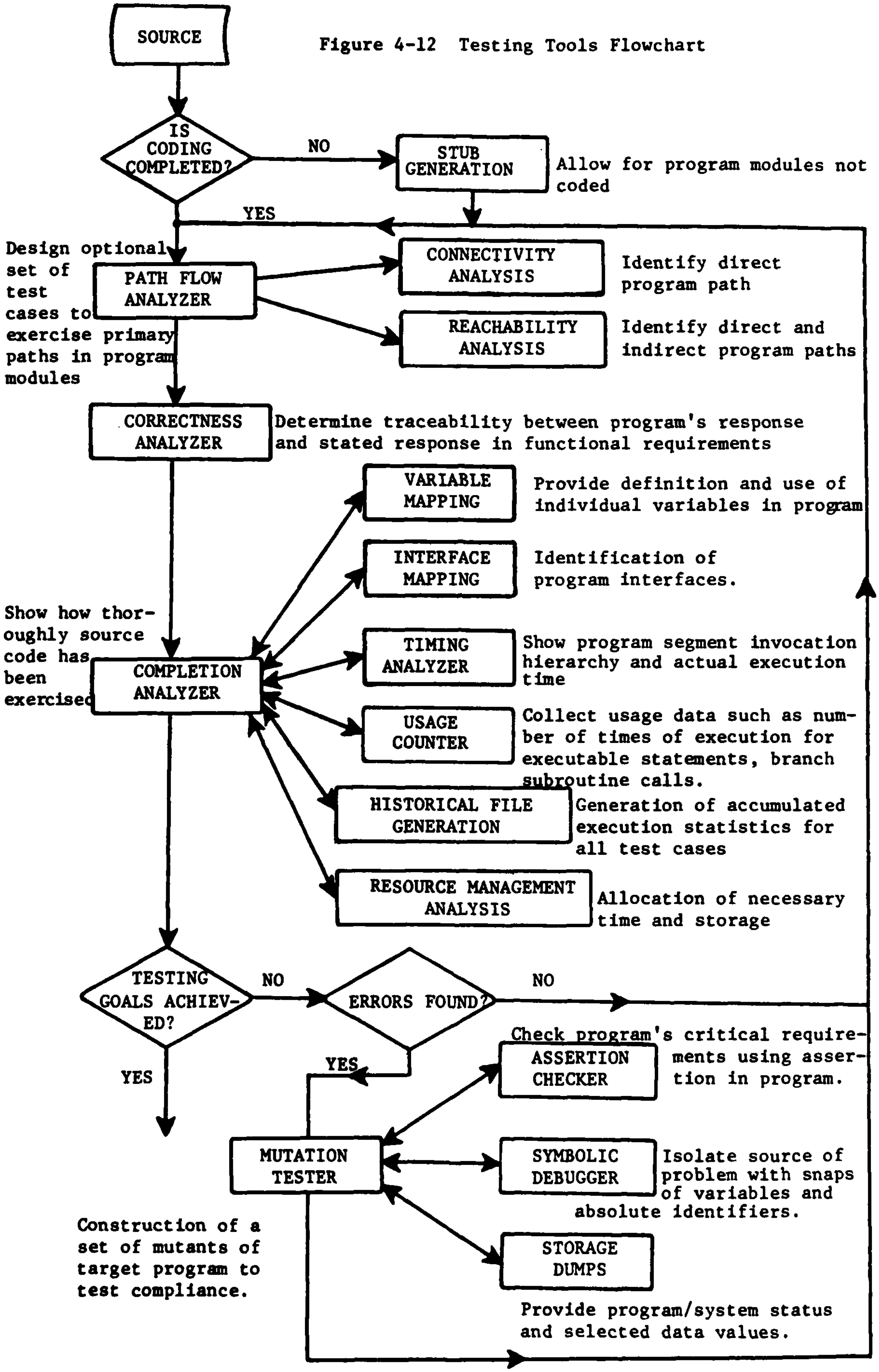


Figure 4-11

Figure 4-12 Testing Tools Flowchart


SOURCE
IS CODING COMPLETED?
NO
STUB GENERATION
Allow for program modules not coded
YES
Design optional set of test cases to exercise primary paths in program modules
PATH FLOW ANALYZER
CONNECTIVITY ANALYSIS
Identify direct program path
REACHABILITY ANALYSIS
Identify direct and indirect program paths
CORRECTNESS ANALYZER
Determine traceability between program's response and stated response in functional requirements
VARIABLE MAPPING
Provide definition and use of individual variables in program
INTERFACE MAPPING
Identification of program interfaces.
Show how thoroughly source code has been exercised
COMPLETION ANALYZER
TIMING ANALYZER
Show program segment invocation hierarchy and actual execution time
USAGE COUNTER
Collect usage data such as number of times of execution for executable statements, branch subroutine calls.
HISTORICAL FILE GENERATION
Generation of accumulated execution statistics for all test cases
RESOURCE MANAGEMENT ANALYSIS
Allocation of necessary time and storage
TESTING GOALS ACHIEVED?
NO
ERRORS FOUND?
NO
YES
YES
Check program's critical requirements using assertion in program.
ASSERTION CHECKER
MUTATION TESTER
SYMBOLIC DEBUGGER
Isolate source of problem with snaps of variables and absolute identifiers.
Construction of a set of mutants of target program to test compliance.
STORAGE DUMPS
Provide program/system status and selected data values.

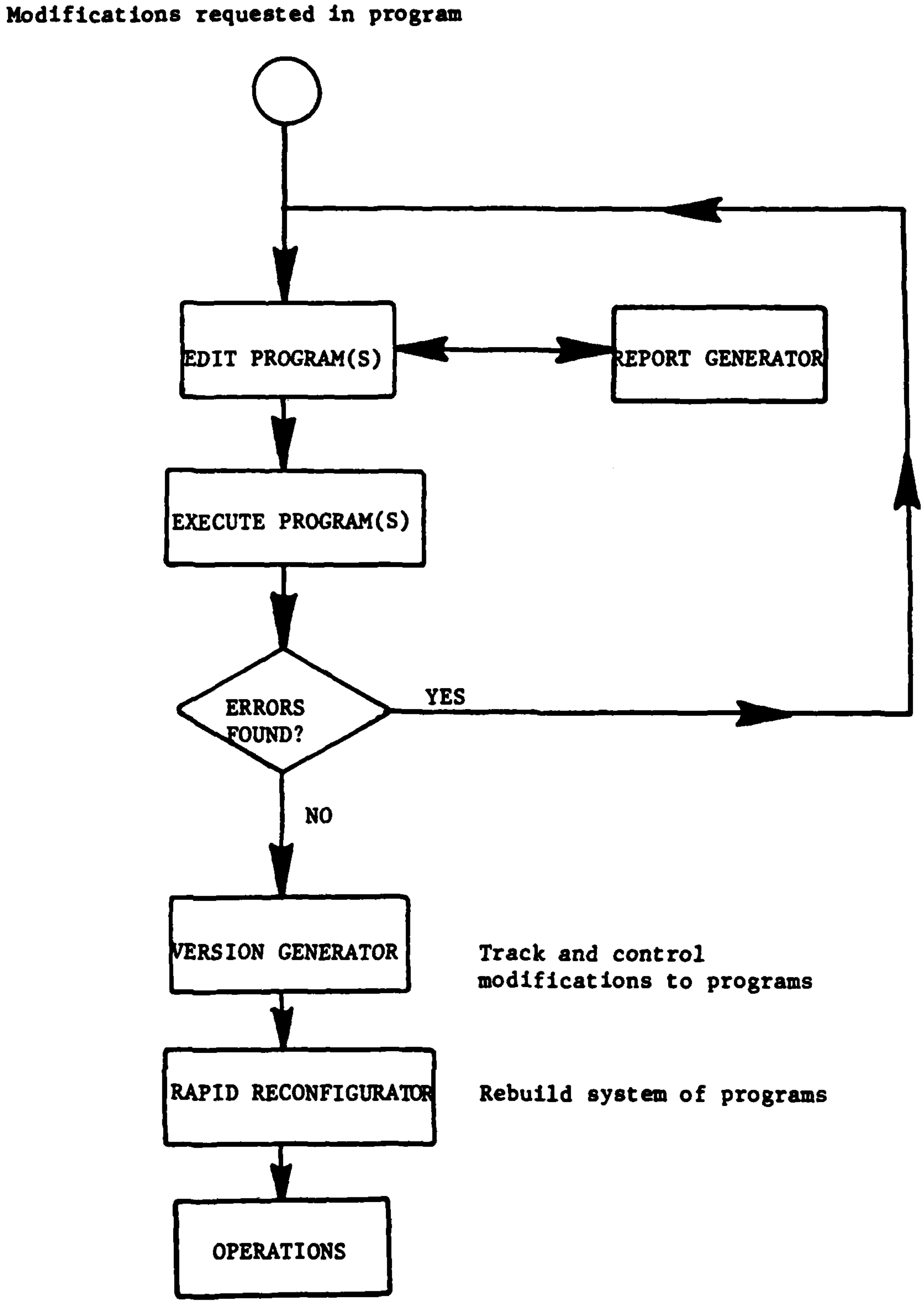


Figure 4-13 Maintenance Tools Flowchart

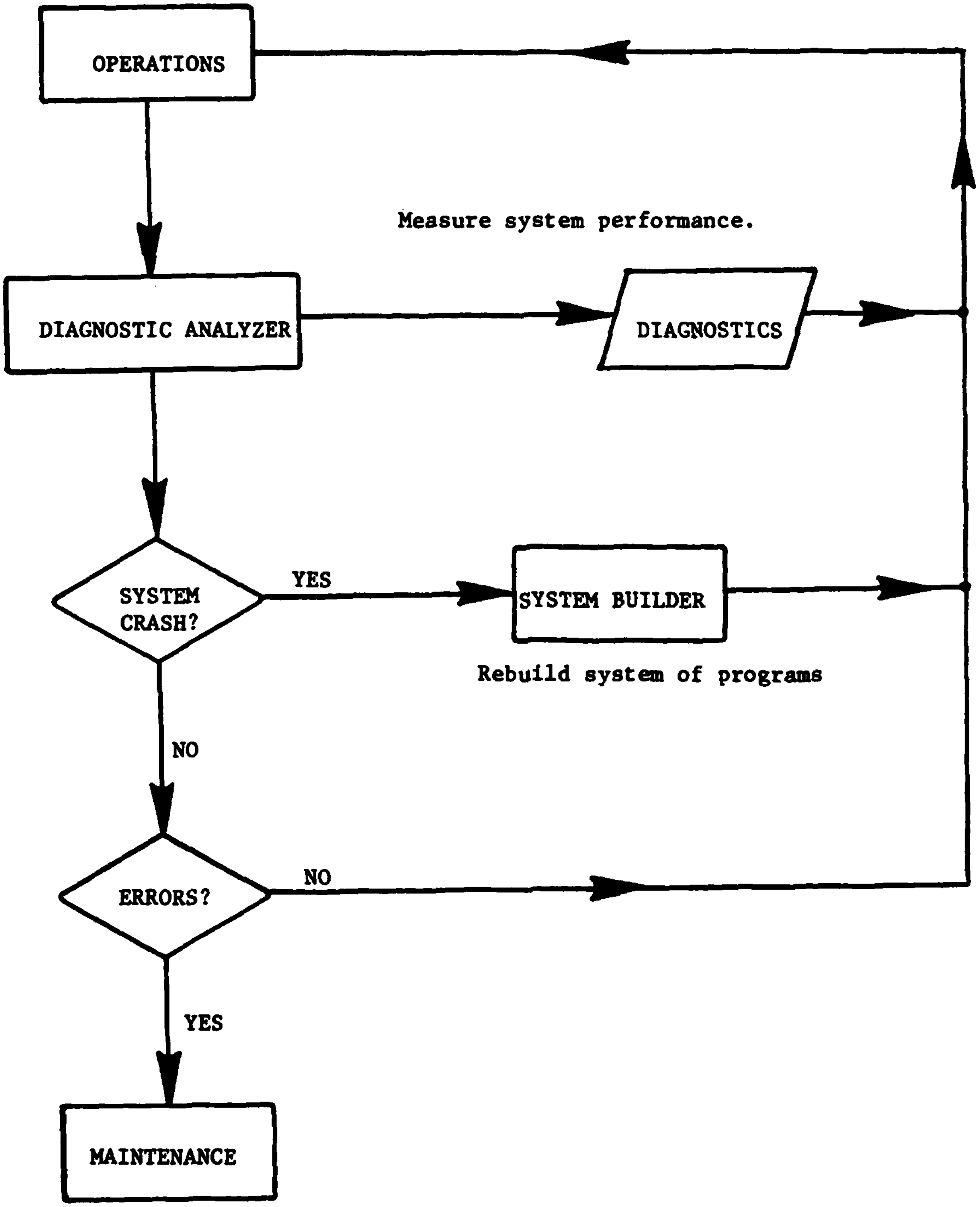


Figure 4-14 Operations Tools Flowchart

4.4 Near-Term Generic Tools Conclusion

In this section, conclusions are drawn based on Tables 4-1 and 4-2. Each life cycle phase was examined to determine whether any criterion applicable to that phase was not enhanced by any of the tools assigned to that phase. These criteria were then used to determine areas where further tool development is needed. The tables were also examined to determine if any criterion which was not applicable to a phase was enhanced by one of the tools assigned to that phase. Tools were grouped according to the phase in which they are more frequently used; however, some tools are used in other phases. For example, the checkout compiler was assigned to the coding phase with the other language tools, but its use overlaps into the testing phase. Because of these overlaps, criteria which were not applicable to a particular life cycle phase may be enhanced by tools assigned to that phase.

In the first life cycle phase, requirements, all applicable criteria are covered except control of data access and independence. It may be that these are difficult criteria to enhance at this early stage. This would certainly seem to be true of independence, but some sort of tool for evaluating the requirements for control of data access would seem to be needed.

The design phase is poverty-stricken with regard to tools. The only tools which exist enhance clarity of documentation. This would be a fertile area for the development of tools to automate the many design methodologies that exist.

The tools available for the coding phase do not address several criteria. These criteria are traceability, functional generality, independence, clarity, virtuality, and distributedness. Major emphasis should be placed on traceability and distribut^dness for new tool development.

The coverage of the testing phase shows the fact that most of the emphasis to date in tool development has been on testing tools. All criteria appropriate to this phase have been addressed.

There are eight applicable criteria which were not addressed for the maintenance phase. The basic problem in maintenance is to retain good qualities already present in the software. Those criteria not addressed are: consistency, modularity, functional generality, instrumentation, independence, compliance, virtuality, and distributedness. This lack of coverage is bad because of the fact that experience shows that approximately 75% of the budget spent on any piece of software is spent in the maintenance phase. The situation is somewhat ameliorated by the fact that many of the tools assigned to other phases can be applied during the maintenance phase.

A reasonable conclusion to be reached from this is that the present emphasis in tool development needs to be changed. The testing phase is probably sufficiently covered, but more research and development needs to be spent on designing tools for the design and maintenance phases. In addition, the criterion of independence is not addressed in any phase; therefore, it looms as a potential area for exploration and development.

5.0 Far-Term Generic Techniques

Distributed processing systems are best supported by an integrated software support environment. Much of the current development effort within the Department of Defense is directed toward such an environment. The subsequent cost savings provide justification for many on-going projects, e.g., the Ada Program Support Environment and various Integrated Software Support Environments. Present emphasis is on implementation as quickly as it can be achieved. However, this overlooks the more subtle long range impact such implementation will have. Why implement integrated software support environments in the first place? Are cost savings alone enough to justify them? Do they provide value-added to the computer users? These and a host of other questions are not addressed. Problems are currently arising within these computer user communities which should be addressed. One such problem concerns the almost apocryphal aspects behind knowledge based systems. An increasing number of computer users are justifying all sorts of new data bases based upon an enhancement to a knowledge based system. Of course a precise definition of what is meant by knowledge based system is usually not addressed. As these undefined levels of expectation become more common within the user community, the relationship of integrated software support environments to knowledge based systems becomes increasingly important. Once implemented, the contribution to specifically defined knowledge based systems by integrated software support environments must be made explicit. This contribution is a function of the emerging role of artificial intelligence. Actual computer intelligence will become evident within the databases and operations of the computer environment. In large part, the success of distributed processing systems will be circumscribed by their ability to contribute to the knowledge based systems of users. The bottom line remains functionality, and what users want is functional knowledge based systems. Consequently, the generic techniques required by distributed processing systems of the future concern knowledge based functionality. Tools not presently envisioned will be required. ISSEs will not be enough. Intelligent and adaptive integrated software support environments will be required. Under present circumstances, the distributed processing environment is complex. Under such future requirements, the complexity multiplies itself. Techniques which will become the tools of tomorrow are going to require artificial intelligence (AI). Although the present only hints of the future, the following observations concerning AI are evident.

The prime far-term generic tool and technique for highly complex systems, including distributed processing systems, will be the application of artificial intelligence. Intelligent components will be a feature of both developmental support systems and the operational systems. Such components will thus support the total software life cycle as well as serve to mediate the complexity of distributed system functions. The most persuasive rationale for the appropriateness of AI is precisely in the potential for managing complexity.

The overall integrating concept here is that of a knowledge based system (KBS). The realization of a KBS is of course somewhat different in the two application areas under discussion (support environments and operational systems). In support (host) environments, the KBS is the foundation of the "intelligent programmer's assistant". In operational (target) systems, intelligence and its supporting knowledge bases function as features of operating systems and data base management. Thus in distributed processing, this knowledge and intelligence will itself be distributed as a system component. Of course, the design of intelligent systems may itself by carried out on an intelligent development system.

Definitions of artificial intelligence usually emphasize the emulation of human cognitive abilities in such tasks as problem solving, symbol manipulation, and operations on incomplete or inaccurate information. Implicit in this emphasis is the assumption of the (metaphorical) ability to "understand". Understanding, in turn, is dependent on an appropriate and adequate representation of knowledge.

The types of knowledge represented and manipulated in a KBS will vary according to whether the KBS pertains to a support environment or to an operational system (though there may be overlap in the content of the two types). A support environment will optimally include detailed knowledge of the application domain(s) (and will include a means for acquiring knowledge about application domains). The design target may be a total system design including hardware/software partitioning, or it may be an application program for an existing target system. In the latter case, knowledge of the target configuration would be part of the knowledge base.

Further, an intelligent support system will have a sophisticated understanding of the application programming language. Any compiler for the language will of course have "knowledge" (but little "understanding") of the syntax and semantics of the language. Various types and levels of intelligence are candidates for incorporation into an intelligent compiler (embodying a knowledge of the pragmatics of the language) and an associated compile-time (and run-time) debugger. (Some of these will be discussed in more detail below.) Understanding of the language (and of the target system) will be useful in other development tools as well, such as a requirements analyzer which outputs source code text. Here the understanding comes into play in selection of appropriate language constructs and facilities. For example, an intelligent encapsulation mechanism (e.g., for packages in Ada) could define classes of objects to be packaged together, using heuristics guided by knowledge of the language rationale, the application domain, and measures of software quality. If low-level objects are specified in a requirements language, this automatic package definition can be viewed as the generation of a high-level, more abstract object.

As indicated above, the representation of knowledge is a fundamentally important issue in KBS development. Representation schemes include production rules (a set of conditions together with a conclusion, perhaps with an attached confidence level), frames, and scripts. Detection of true conditions in production rules may involve heuristic evaluation. An important aspect of knowledge representation is the association of teleology (purpose) with raw information. Inclusion of purpose aids an intelligent system in evaluation of the relevance of information to a task or problem.

Intelligence in knowledge-based, expert consulting systems will provide decision aids throughout the system life cycle. This will be valuable in any activity where the consequence of experimental adjustment of system parameters needs to be evaluated. An example is impact analysis of requirements or design changes, where what is desired is an evaluation of the severity of a proposed change. AI techniques can be used to search the knowledge base for relevant conceptual connections. Efficient heuristics could make feasible an interactive dialogue with the decision-maker. The search process in this example may be sufficient but not exhaustive. Once a change decision has been made a detailed analysis of affected system entities will be necessary. In this case, search will be exhaustive, but can be intelligently guided to avoid blind search.

A further desirable component of the knowledge base for decision aids will be the inclusion of quantitative and qualitative software metrics. This will enable an intelligent dialogue with program designers in which alternative design features can be evaluated against criteria of goodness. The decisions of the designer can feed back into the knowledge base so that future decisions can be more fully automated. A mature design knowledge base will be useful in rapid system prototyping.

Support tools of the future will be used in designing and implementing target systems with artificially intelligent components. A principal motivation for the use of AI techniques in distributed processing is the management of the inherent complexity of such systems. Since intelligent support systems and intelligent components of target systems will use the same generic technology, it is less expedient to detail the intelligent functions in the target systems. Likely there will be beneficial technology exchange in AI between tool development activity and applications program techniques. For the sake of completeness, we will lastly consider some important potential applications of AI in the target systems.

Local operating systems will require intelligence to direct decisions based on incomplete state information for resource management and for recovery. Intelligence can be applied to nondeterministic scheduling and task allocation in concurrent software. Application algorithms may be automatically partitioned and distributed to separate processors for parallel execution.

There will be intelligent components of data base management. Intelligent retrieval will make use of inferencing capacity and of strategies for merging schemas of distributed data bases. An emerging concept is that of active data bases, which will report evolving patterns of interest. Finally, natural language processing will be a feature of the user interface.

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
APSE	Ada Programming Support Environment
COS	Constituent Operating System
DBMS	Data Base Management System
DoD	Department of Defense
DOS	Distributed Operating System
IA	Interconnect Architectures
IEEE	Institute of Electrical and Electronic Engineers
I/O	Input/Output
ISSE	Integrated Software Support Environment
KAPSE	Kernel Ada Program Support Environment
KBS	Knowledge Based System
KIT	KAPSE Interface Team
MAPSE	Minimal Ada Program Support Environment
MCMD	Multi-Center, Multi-Drop
MDMS	Multi-Center, Multi-Star
MST	Minimal Spanning Tree
PDL	Program Design Language
RADC	Rome Air Development Center
ROM	Read Only Memory
SCMD	Single-Center, Multi-Drop
SCSS	Single-Center, Single-Star

Bibliography

1. Andrews, D. M. and Melton, R. A., FAVS: FORTRAN Automated Verification System User's Manual, General Research Corp. Report CR-1-754/1, April, 1980.

2. Baklovich, E., Decentralized Systems. Computer Science Technical Report, University of Connecticut, AD/A099 195, Storrs, Connecticut, 1980.

3. Barr, A. and Feigenbaum, E. A., editors, The Handbook of Artificial Intelligence, Los Altos, CA, William Kaufman, 1981.

4. Benoit, John W. and Selander, J. Michael, Knowledge-Based Systems as Command Decision Aids, First U.S. Army Conference on Knowledge-Based Systems for C^3I, 1981.

5. Clark, Lori A. et al, Toward Feedback-Directed Development of Complex Software Systems, University of Massachusetts, Amherst, Massachusetts.

6. Cook, R. P., A Review of the Stoneman APSE Specification, Consulting Report, June, 1982.

7. Cook, R. P., How To Write a Distributed Program, Consulting Report, June, 1982.

8. Cook, R. P., Kernel Design for Concurrent Programming, Consulting Report, June, 1982.

9. Daley, P., Modeling of Distributed Command/Communication/Control Intelligence Systems, RADC Distributed Processing Technology Exchange, May, 1982.

10. Department of Defense, Reference Manual for the Ada Programming Language, July, 1980.

11. Donahoo, J. D. and Swearinger, D., A Review of Software Technology, RADC-TR-80-13, February, 1980.

12. Drazovich, Robert J. and Payne, J. Roland, Artificial Intelligence Approaches to Information Fusion, First U.S. Army Conference on Knowledge-Based Systems for C^3I, 1981.

13. Enslow, P., Performance of Distributed and Decentralized Control Models for Fully Distributed Processing Systems, RADC-TR-82-105, May, 1982.

14. Enslow, P., Support for Loosely-Coupled Distributed Processing Systems, RADC Distributed Processing Technology Exchange, May, 1982.

15. Feng, Tse-Yun and Wu, Chuan-lin, Interconnection Networks in Multiple-Processor Systems, RADC-TR-79-304, December, 1979.

16. Findler, N. V., editor, Associative Networks: The Representation and Use of Knowledge by Computers, New York, Academic Press, 1979.

17. First U.S. Army Conference on "Knowledge-Based Systems for C^3I", Ft. Leavenworth, Kansas, 4-5 November, 1981.

18. Forsdick, Harry C., et al, Distributed Operating System Design Study, RADC-TR-81-384, January, 1982.

19. Fortier, P. J. and Leary, R. G., A General Simulation Model for the Evaluation of Distributed Processing Systems, Annual Simulation Symposium, November, 1981.

20. Gannon, C. and Brooks, N. B., JOVIAL J73 Automated Verification System Functional Description, General Research Corp. Report CR-1-947, March, 1980.

21. Giese, C., Research and Development Plan for Ada Target Machine Operating System (ATMOS) for the Ada Bare Target Machine, AJPO/U.S. Army AIRMICS, April, 1982.

22. Gorney, L., Queueing Theory: A Problem Solving Approach, Petrocelli Books, 1982.

23. Green, Cordell, A Knowledge-Based Approach to Rapid Prototyping, Software Engineering Symposium: Rapid Prototyping, 1982.

24. Hayes-Roth, Frederick, Artificial Intelligence and Expert Systems, A Tutorial, First U.S. Army Conference on Knowledge-Based Systems for C^3I, 1981.

25. Jensen, D., Decentralized System Control, RADC Distributed Processing Technology Exchange, May, 1982.

26. Jensen, E. Douglas, The ARCHONS Project, RADC Distributed Processing Exchange, October, 1981.

27. Joobbani, R. and Siewiorek, D. P., Reliability Modeling of Multiprocessor Architectures, Carnegie-Mellon University, Pittsburg, PA, 1979.

28. Kemp, G. H., Debugging Embedded Computer Programs, GDPD Technical Memorandum, March, 1980.

29. McCall, J. A. and Matsumoto, M. T., Software Quality Metrics Enhancements, RADC-TR-80-109, Volumes I and II, April, 1980.

30. Melton, R., Grunburg, G. and Sharp, M., COBOL Automated Verification System: Study Phase, RADC-TR-81-11, March,1981.

31. Post, J., Quality Metrics for Distributed Systems, RADC Distributed Processing Technology Exchange, May, 1982.

32. Reinstein, H. C. and Hollander, C. R., A Knowledge-Based Approach to Application Development for Non-Programmers, IBM Palo Alto Scientific Center, July, 1979.

33. Saponas, T. G., Distributed and Decentralized Control in Fully Distributed Processing Systems, GIT-ITC-81/18, December, 1981.

34. Sharma, R. L., deSousa, P. J. T., and Ingle, A. D., Network Systems, Van Nostrand Reinhold Data Processing Services, 1982.

35. Sharp, M., Melton, R. and Greenburg, G., COBOL Automated Verification System Functional Description, General Research Corp. Report CR-2-970, November, 1980.

36. Stenning, V., et al, The Ada Environment: A Perspective, Computer, June, 1981.

37. Tanenbaum, A. S., Computer Networks, Prentice-Hall, Englewood Cliffs, NJ, 1981.

38. Taylor, R. N., Complexity of Analyzing the Structure of Concurrent Programs, University of Victoria Department of Computer Science, 1981.

39. Taylor, R. N. and Osterweil, L. J., Anomaly Detection in Concurrent Software by Static Data Flow Analysis, IEEE Transactions on Software Engineering, Volume 6, 1980.

40. Thomas, R., Distributed Personal Computer-Based Information Systems, RADC Distributed Processing Technology Exchange, May, 1982.

41. Wolfe, M. I., et al, The Ada Language System, Computer, June, 1981.

42. Ziegler, K., A Distributed Information System Study, IBM System Journal, Volume 18, Number 3, 1979.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.